

# 课程目标

---

- ✓ 理解算法的**时间-空间复杂度**的分析方法
- ✓ 理解和掌握算法中的**递归、分治法、贪心算法、图、动态规划**等基本思想
- ✓ 能够**运用**经典算法的思想和方法
- ✓ 能够尝试提出**改进**解决方案的思路及方法
- ✓ 针对一定复杂程度的问题求解，**设计并实现**算法流程



# 重要不重要？

---

计算机专业的学生应掌握的技能

- 编程语言？
- 算法？
- 程序设计大赛
- 找工作面试
- 研究生及博士入学考试科目

# 如何学好？

学习方法：

- 掌握算法设计的核心思想（基础）
- 注重编程实现（提高）

You Are Supposed to Be:

Cool minded

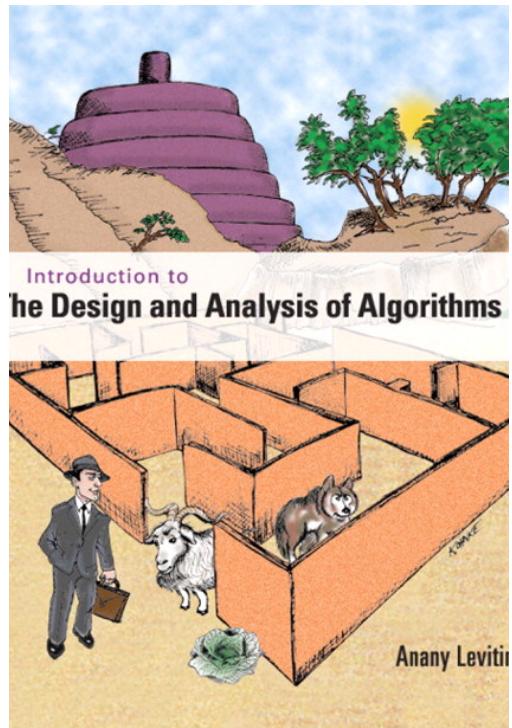
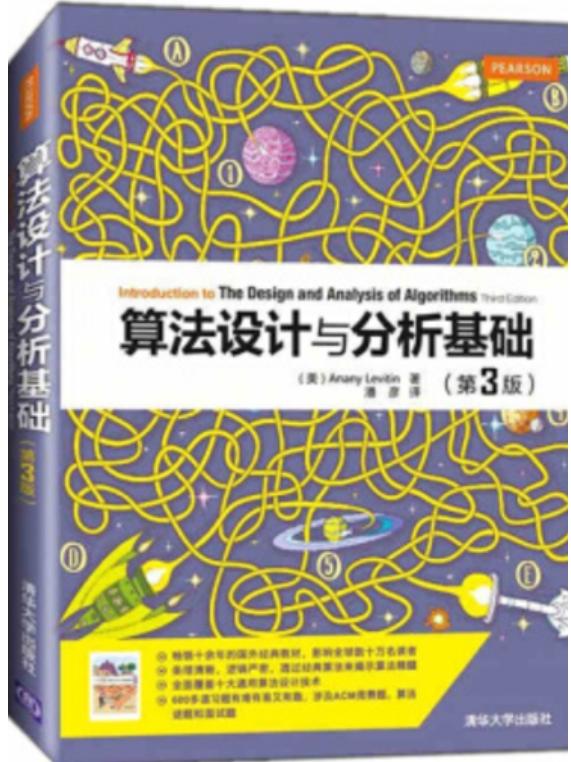
Self motivated

Programming addicted





# 教材



**Introduction to design and analysis of  
Algorithms (Third Edition)**

**Anany Levitin**

# 参考书

## Introduction to Algorithms

T. H. Cormen, C. E. Leiserson, R. L. Rivest (2002, Turing Award)

The MIT Press

## Algorithms

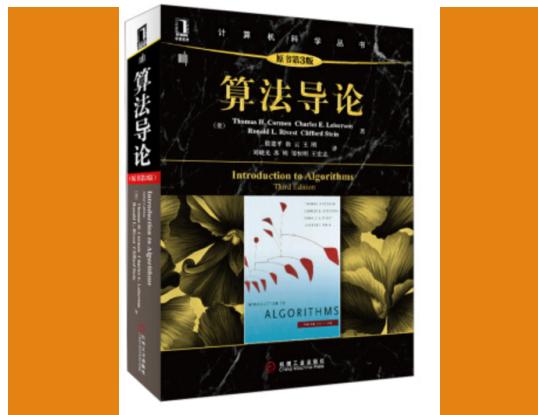
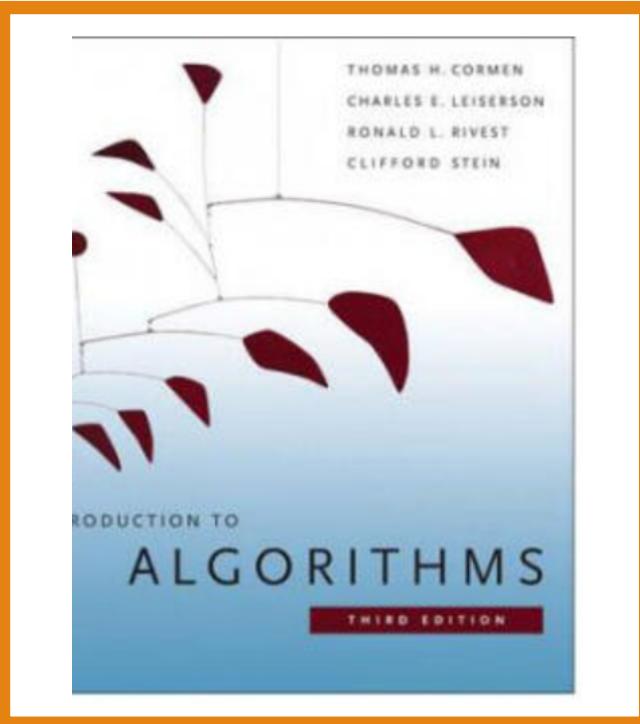
R. Sedgewick

## The Art of Computer Programming

Donald E. Knuth, 1974, Turing Award

----Bill Gates: “如果你认为你是一名真正优秀的程序员，请读Knuth的《计算机程序设计艺术》，如果你能读懂整套书的话，请给我发一份你的简历。”

《计算机算法设计与分析》 王晓东



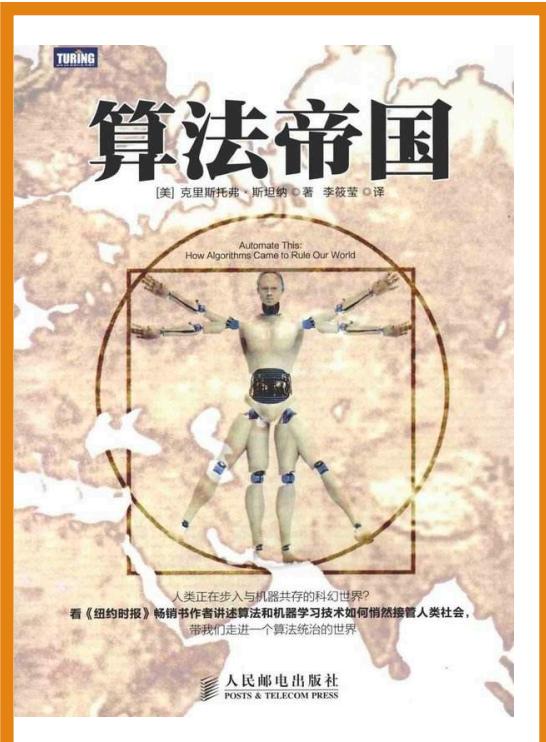
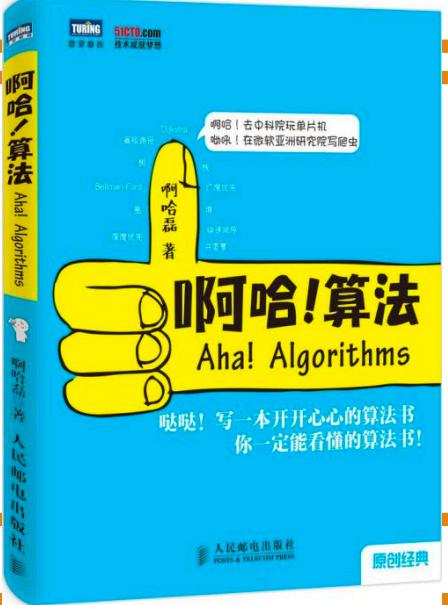
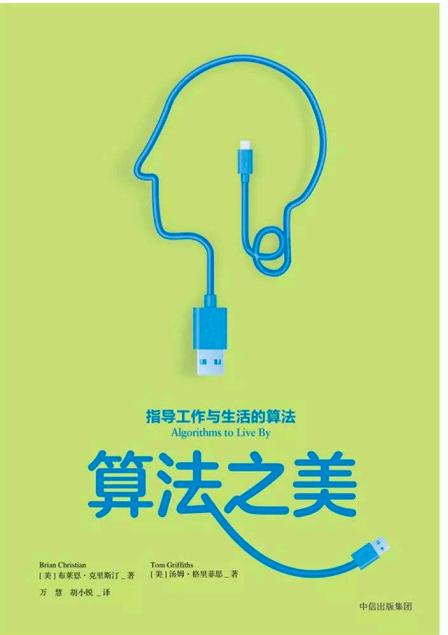
# 课程资源

- 公开课

- 算法设计与分析（北京大学-屈婉玲）  
<https://www.coursera.org/learn/algorithms>
- 算法导论（麻省理工）  
<http://open.163.com/special/opencourse/algorithms.html>
- 算法（普林斯顿大学）  
<https://www.coursera.org/learn/algorithms-part1>  
<https://www.coursera.org/learn/algorithms-part2>

- 专项课程

- 程序设计与算法（北京大学）  
[www.coursera.org/specializations/biancheng-suanfa](http://www.coursera.org/specializations/biancheng-suanfa)
- 算法（斯坦福大学）  
[www.coursera.org/specializations/algorithms](http://www.coursera.org/specializations/algorithms)





# 编程资源

---

- 在线题库

- Hackerrank: <https://www.hackerrank.com/>
- Topcoder: <https://www.topcoder.com/>
- Geeksforgeeks: <https://www.geeksforgeeks.org/>
- Codeforces: <http://codeforces.com/>
- Lintcode: <https://www.lintcode.com/>
- Leetcode: <https://leetcode.com/>
- POJ-北大: <http://poj.org/>
- HDU-杭电: [acm.hdu.edu.cn/](http://acm.hdu.edu.cn/)



# 算法设计与分析基础

---

PART1: 算法概述



# 主要内容

---

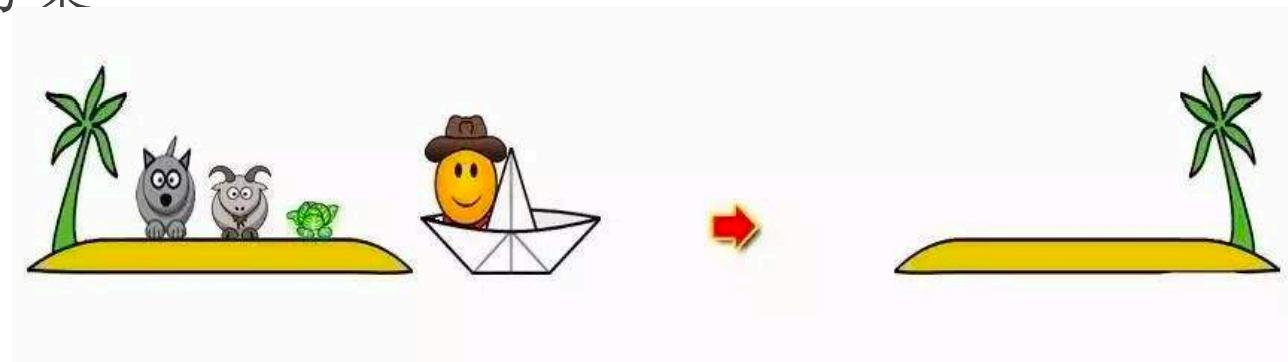
- 什么是算法？
- 重要问题类型
- 基本数据结构
- 算法效率分析

# 趣味问题——农夫过河

## 1、问题描述

一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸。他要这些东西全部运到北岸。他面前只有一条小船，船只能容下他和一件物品，另外只有农夫才能撑船。如果农夫在场，则狼不能吃羊，羊不能吃白菜，否则狼会吃羊，羊会吃白菜，所以农夫不能留下羊和白菜自己离开，也不能留下狼和羊自己离开，而狼不吃白菜。

请求出农夫将所有的东西运过河的方案



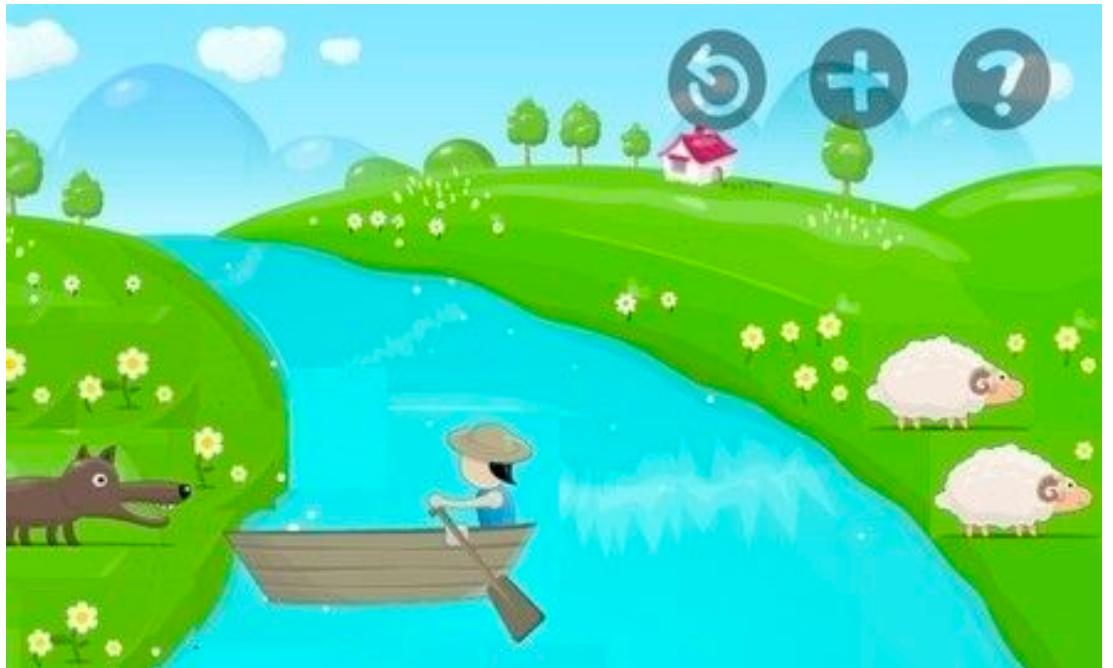
# 趣味问题——农夫过河

## 2、解决步骤

- 带羊到对岸，返回；
- 带菜到对岸，把羊带回；
- 带狼到对岸，返回；
- 带羊到对岸。

## 3、计算机算法解决思路

图算法（深度优先或广度优先搜索）





# 最大公约数问题

最大公约数 (greatest common divider) 的定义:

能够同时整除 (即余数是0) 两个不全为0非负整数的最大正整数。

## 最大公约数问题

求两个不全为0的非负整数 (m和n) 的最大公约数 (记为gcd(m, n)) 。

### 例子

$$\text{gcd}(5, 0) = 5$$

$$\text{gcd}(15, 5) = 5$$

$$\text{gcd}(60, 24) = 12$$

# 方法一：欧几里得算法（辗转相除法）

## 算法要点

- $gcd(m, n) = gcd(n, m \bmod n)$ , 其中  $m \bmod n$  为  $m$  除  $n$  后的余数。
- $gcd(m, 0) = m$

## 计算 $gcd(m, n)$ 的欧几里得算法

- ① 如果  $n = 0$ , 返回  $m$  的值作为结果, 过程结束; 否则, 进入第2步;
- ②  $m$  除以  $n$ , 得到余数  $r$ ;
- ③ 将  $n$  的值赋给  $m$ , 将  $r$  的值赋给  $n$ , 返回第1步;

## 计算 $gcd(60, 24)$

$$gcd(60, 24)$$

# 练习: 利用欧几里得算法求 $\gcd(31415, 14124)$

答案

$$\begin{aligned}\gcd(31415, 14124) &= \gcd(14124, 31415 \bmod 14124) = \gcd(14124, 3131) \\&= \gcd(3131, 14124 \bmod 3131) = \gcd(3131, 1618) \\&= \gcd(1618, 3131 \bmod 1618) = \gcd(1618, 1513) \\&= \gcd(1513, 1618 \bmod 1513) = \gcd(1513, 105) \\&= \gcd(105, 1513 \bmod 105) = \gcd(105, 43) \\&= \gcd(43, 105 \bmod 43) = \gcd(43, 19) \\&= \gcd(19, 43 \bmod 19) = \gcd(19, 5) \\&= \gcd(5, 19 \bmod 5) = \gcd(5, 4) \\&= \gcd(4, 5 \bmod 4) = \gcd(4, 1) \\&= \gcd(1, 4 \bmod 1) = \gcd(1, 0)\end{aligned}$$



如何描述此算法?

# 方法二：连续整数检测法

## 算法要点

基于最大公约数的定义， $m$ 和 $n$ 的最大公约数是能够整除它们的最大正整数，因此可以从 $m$ 和 $n$ 中较小的数逐步减1寻找最大公约数。

### 计算 $gcd(m, n)$ 的连续整数检测算法

- ① 将 $\min(m, n)$ 的值赋给 $t$ ；
- ②  $m$ 除以 $t$ ，如果余数为0，进入第3步，否则进入第4步；
- ③  $n$ 除以 $t$ ，如果余数为0，返回 $t$ 的值，否则进入第4步；
- ④ 将 $t$ 的值减1，返回第2步；

## 注意

当算法的一个输入是0的时候，连续整数检测算法得到的结果是错误的。因此，在算法的设计中必须认真、清晰地规定算法的输入的值域。

# 方法三：中学的计算方法

中学时计算 $gcd(m, n)$ 的过程

- ① 找到 $m$ 的所有质因数；
- ② 找到 $n$ 的所有质因数；
- ③ 找出所有的公因数；
- ④ 将找到的公因数相乘，结果即为最大公因数；

实例

$$60 = 2 * 2 * 3 * 5$$

$$24 = 2 * 2 * 2 * 3$$

$$gcd(60, 24) = 2 * 2 * 3 = 12$$

# 方法三：中学的计算方法

## 注意

- 利用上述过程计算  $gcd(31415, 14124)$  有什么问题？
- 在给出具体的质因数计算过程之前，上述方法不能称为一个真正意义上的算法。
- 计算连续质数序列：埃拉托色尼筛选法（课本Page5）

# 算法的定义

## ○ 数学

算法通常是指按照一定规则解决某一类问题的明确和有限的步骤。

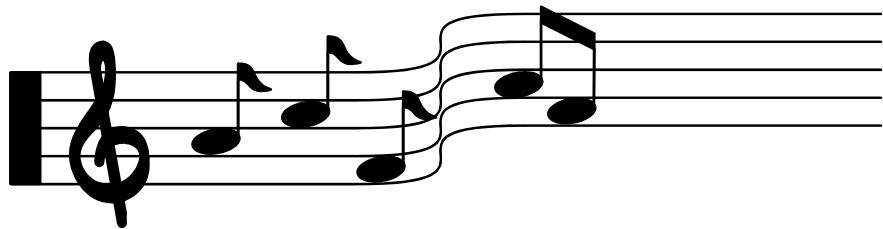


## ○ 广义

算法是完成某项工作的方法和步骤。

菜谱是做菜的算法

歌谱是一首歌曲的算法



## ○ 计算机科学

算法是一系列解决问题的明确指令，也就是说，对于**符合一定规范**的输入，能够在**有限时间**内获得要求的输出。

# 算法的定义

算法是指解决问题的一种方法或一个过程。

算法是若干指令的有穷序列，满足性质：

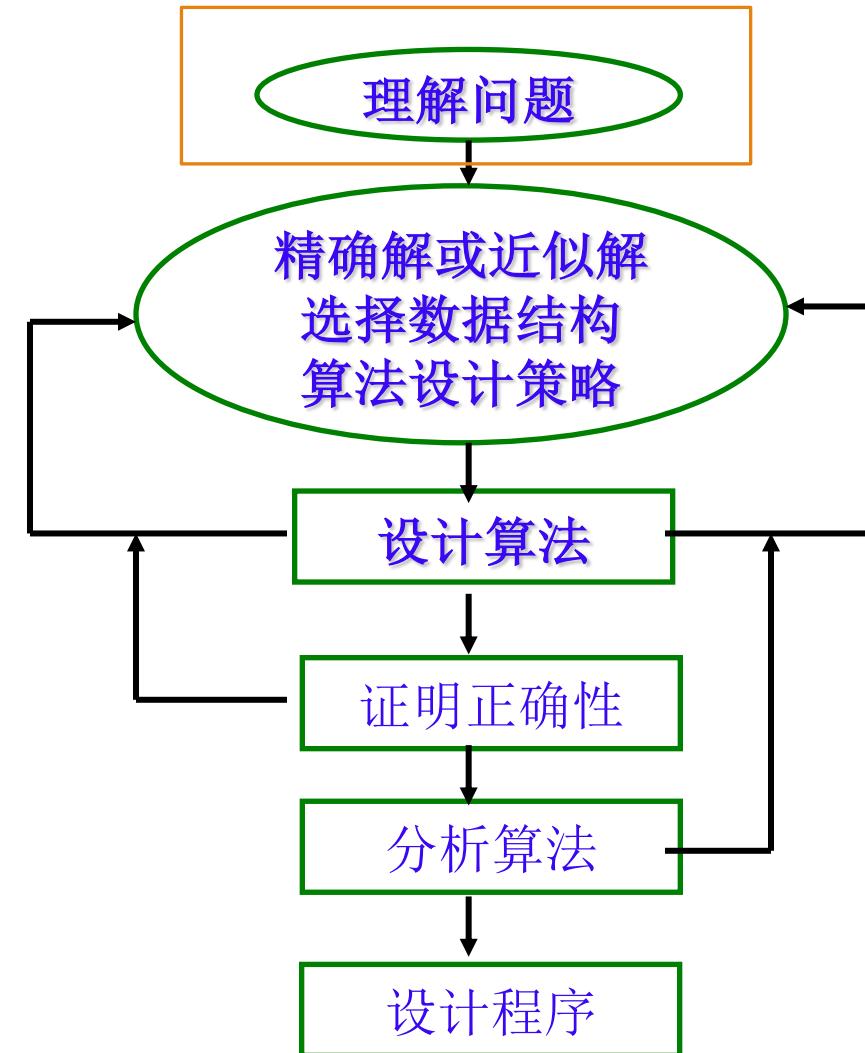
- (1) **输入**：有外部提供的量作为算法的输入。
- (2) **输出**：算法产生至少一个量作为输出。
- (3) **确定性**：组成算法的每条指令是清晰，无歧义的。
- (4) **有限性**：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。

# 程序 (Program)

- 程序是算法用某种程序设计语言的具体实现。
- 程序可以不满足算法的性质 (4)
- 思考：操作系统是算法吗？

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





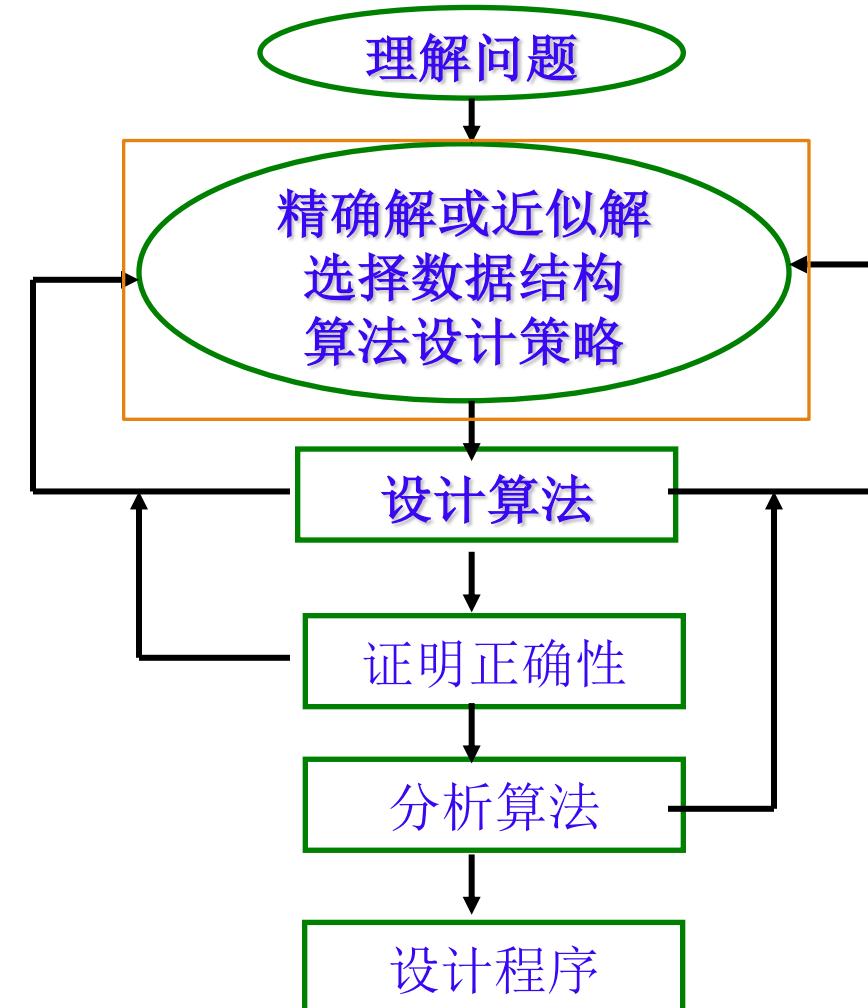
# 理解问题

---

1. 完全理解给定问题
2. 仔细阅读问题描述，并提出问题
3. 先做一些小例子去理解，并且考虑特殊情况
4. 再次提出问题

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





# 确定计算设备的性能

---

## ➤ 顺序算法

- 冯·诺伊曼体系结构 Von Neumann machine
- RAM: 指令依次执行, 一次执行一个操作

## ➤ 并行算法

- 同时执行操作



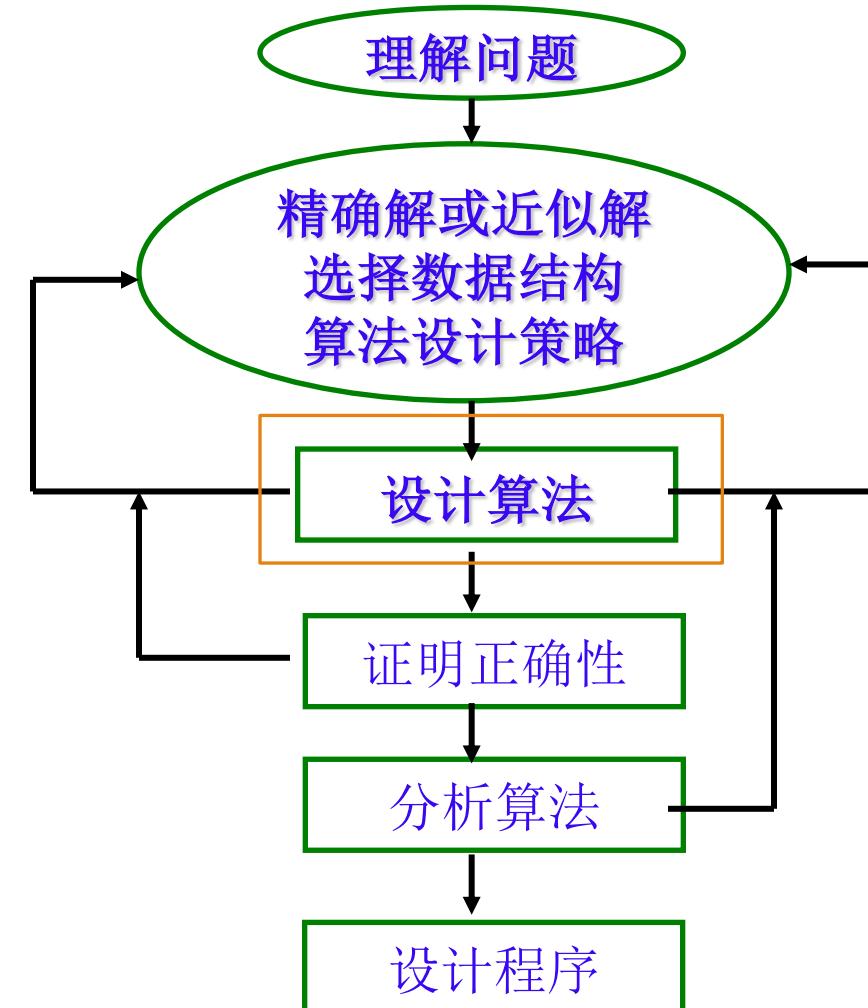
# 选择精确或近似解

---

- 精确解算法
- 近似解算法
- 一些重要的问题对于大多数实例不能求得精确解
- 已知精确解的算法不能在可接受的时间内解决问题

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





# 设计算法和数据结构

---

描述算法的方法：

- 自然语言
- 伪代码 **Pseudocode**
- 介于自然语言和编程语言之间
- 比自然语言更准确
- 没有统一的标准
- 流程图



# 最大公约数问题——欧几里得算法

自然语言 Nature Language:

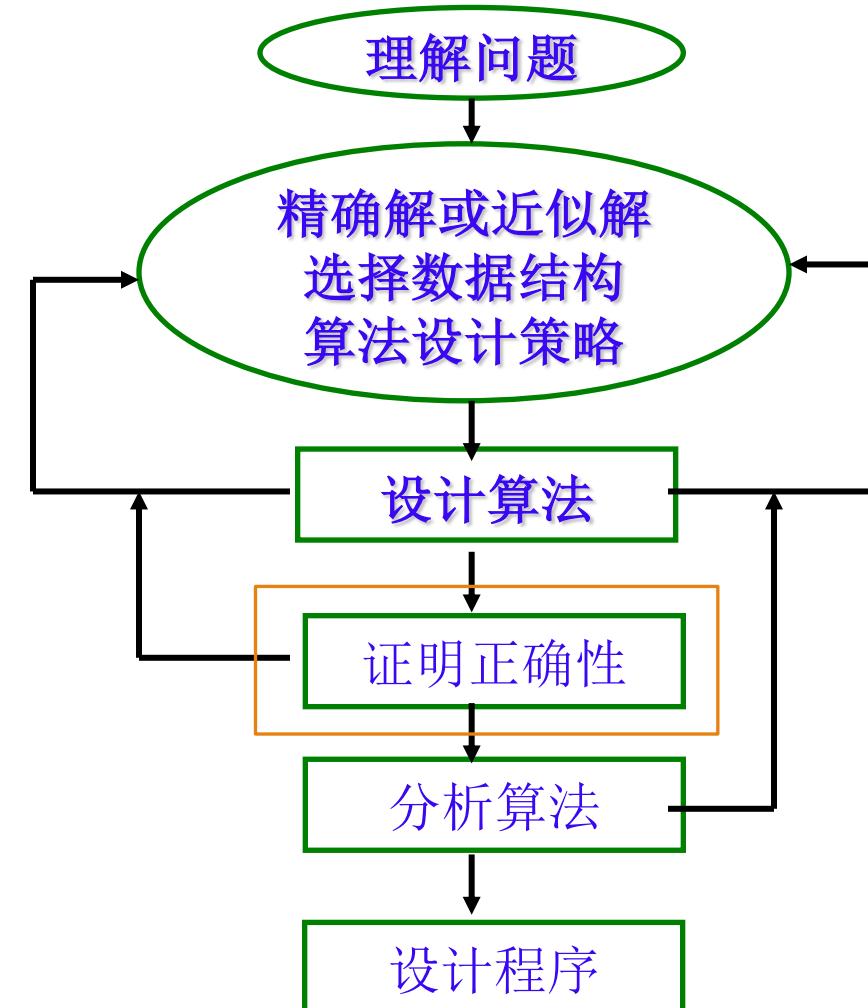
- Step 1 If  $n = 0$ , return  $m$  and stop; otherwise go to Step 2
- Step 2 Divide  $m$  by  $n$  and assign the value of the remainder to  $r$  Step 3 Assign the value of  $n$  to  $m$  and the value of  $r$  to  $n$ . Go to Step

伪代码 Pesudo code:

```
while  $n \neq 0$  do
     $r \leftarrow m \bmod n$ 
     $m \leftarrow n$ 
     $n \leftarrow r$ 
return  $m$ 
```

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





# 算法的正确性证明

**正确性:** 需要证明算法是在**有限时间内**得到**合法输入**的**预期结果**

**数学归纳法:** 证明算法的迭代过程符合这一证明所需要的步骤

**近似算法的正确性:** 算法得到的输出结果误差是否在一个限定范围内

# 算法设计与分析的重要问题类型

- **排序问题**
- **查找问题**
- **图问题:** 主要涉及图的遍历和图的拓扑排序的内容
- **组合问题:** 要求从离散的空间中寻找一个对象,使之满足特定的标准(如满足某种最优化性质)
- **数值问题:** 涉及连续性的数学问题,解空间无穷大,近似解
- **几何问题:** 处理点、线、面这些对象的一些问题

# 算法设计的主要策略

- 蛮力算法(穷举算法):直接对问题进行求解
- 分治算法
- 贪心算法
- 动态规划算法
- 回溯算法
- 分支限界算法
- 随机化算法(概率算法)
- 线性规划算法
- NP完全性理论与近似算法
- 计算智能

# 排序问题

## 问题定义

"排序"就是把一组杂乱的数据按照一定的规律排列起来。

## 为什么需要有序列表？

- 有序列表是所求解问题的输出，如学生成绩排序。
- 在很多其他领域的算法（如几何算法和数据压缩算法），排序被作为一个辅助步骤。

## 排序算法

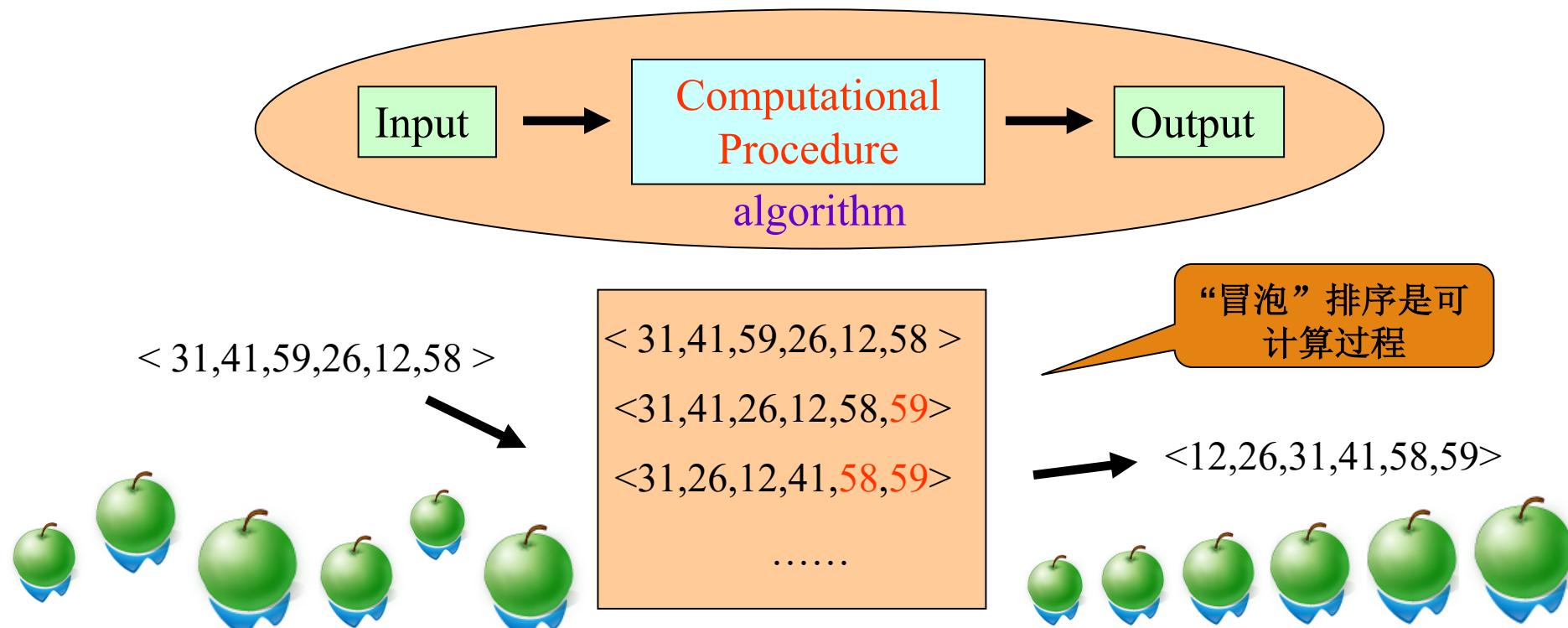
- 选择排序，插入排序，冒泡排序，快速排序，堆排序，二分归并排序，堆排序等。
- 虽然有些算法在平均效率和最差情况下的效率比其他算法都好，但是没有一种算法在任何输入下都是最优的。

# 排序问题

Problem: to sort a sequence of numbers into nondecreasing order

Input: A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$

Output: A permutation (reordering)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$



# 冒泡排序

例子：用冒泡法对8个数排序

排序过程：

- (1) 比较第一个数与第二个数，若为逆序 $a[0]>a[1]$ ，则交换；然后比较第二个数与第三个数；依次类推，直至第 $n-1$ 个数和第 $n$ 个数比较为止——第一趟冒泡排序，结果最大的数被安置在最后一个元素位置上
- (2) 对前 $n-1$ 个数进行第二趟冒泡排序，结果使次大的数被安置在第 $n-1$ 个元素位置
- (3) 重复上述过程，共经过 $n-1$ 趟冒泡排序后，排序结束

n=8

|       |    |    |     |     |     |     |     |     |
|-------|----|----|-----|-----|-----|-----|-----|-----|
| 例     | 38 | 38 | 38  | 38  | 13  | 13  | 13  | 13  |
|       | 49 | 49 | 49  | 13  | 27  | 27  | 27  | 27  |
|       | 65 | 65 | 13  | 27  | 30  | 30  | 30  |     |
|       | 76 | 13 | 27  | 30  | 38  | 38  |     |     |
|       | 13 | 27 | 30  | 49  | 49  |     |     |     |
|       | 27 | 30 | 65  | 65  |     |     |     |     |
|       | 30 | 76 | 76  |     |     |     |     |     |
|       | 97 | 97 | 第一趟 | 第二趟 | 第三趟 | 第四趟 | 第五趟 | 第六趟 |
| 初始关键字 |    |    |     |     |     |     |     | 第七趟 |

输入n 个数给a[1] 到 a[n]

for j=1 to n-1

for i=1 to n-j

真

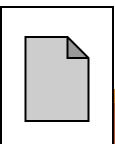
a[i]>a[i+1]

假

a[i]↔a[i+1]

输出a[1] 到 a[n]

```
#include <stdio.h>
main()
{ int a[11],i,j,t;
printf("Input 10 numbers:\n");
for(i=1;i<11;i++)
    scanf("%d",&a[i]);
printf("\n");
for(j=1;j<=9;j++)
    for(i=1;i<=10-j;i++)
        if(a[i]>a[i+1])
            {t=a[i]; a[i]=a[i+1]; a[i+1]=t;}
printf("The sorted numbers:\n");
for(i=1;i<11;i++)
    printf("%d ",a[i]);
}
```



# 选择排序

例子：用选择排序法对n个数排序

排序过程：

- (1) 首先通过 $n-1$ 次比较，从 $n$ 个数中找出最小的，将它与第一个数交换—第一趟选择排序，结果最小的数被安置在第一个元素位置上
- (2) 再通过 $n-2$ 次比较，从剩余的 $n-1$ 个数中找出关键字次小的记录，将它与第二个数交换—第二趟选择排序
- (3) 重复上述过程，共经过 $n-1$ 趟排序后，排序结束

|   |       |     |   |
|---|-------|-----|---|
| 例 | $i=1$ | 初始: | [ 13    38    65    97    76    49    27 ]  |
|   |       |     | $\downarrow k$<br>$\downarrow k$<br>$\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$ |
|   | $i=2$ | 一趟: | 13      27    65    97    76    49    38 ]  |
|   |       |     | $\downarrow k$<br>$\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$                                |
|   |       | 二趟: | 13    27    [ 65    97    76    49    38 ]  |
|   |       |     | $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$  |
|   |       | 三趟: | 13    27    38    [ 97    76    49    65 ]  |
|   |       |     | $\uparrow j$ $\uparrow j$ $\uparrow j$ $\uparrow j$   |
|   |       | 四趟: | 13    27    38    49    [ 76    97    65 ]  |
|   |       |     | $\uparrow j$ $\uparrow j$ $\uparrow j$  |
|   |       | 五趟: | 13    27    38    49    65    [ 97    76 ]  |
|   |       |     | $\uparrow j$ $\uparrow j$   |
|   |       | 六趟: | 13    27    38    49    65    76    [ 97 ]  |

输入n个数给a[1] 到 a[n]

for i=1 to n-1

k=i

for j=i+1 to n

真

a[j]<a[k]

假

k=j

真

i != k

假

a[i]↔a[k]

输出a[1] 到 a[n]

```
#include <stdio.h>
```

```
main()
```

```
{ int a[11],i,j,k,x;  
printf("Input 10 numbers:\n");  
for(i=1;i<11;i++)  
    scanf("%d",&a[i]);  
printf("\n");  
for(i=1;i<10;i++)  
{ k=i;  
    for(j=i+1;j<=10;j++)  
        if(a[j]<a[k]) k=j;  
    if(i!=k)  
    { x=a[i]; a[i]=a[k]; a[k]=x; }  
}  
printf("The sorted numbers:\n");  
for(i=1;i<11;i++)  
    printf("%d ",a[i]);  
}
```

# Identifying Genes in Human DNA (基因识别)

- Identifying all the 100,000 genes in human DNA
  - ◆ determining the **sequences** of the 3 billion( $10^9$ ) chemical base pairs that make up human DNA.

(30亿组化学基对组成人类DNA，如何界定这些序列，从而进行基因识别)



- Computer: 3G Hz CPU,  $3 \times 10^9$ B/s, suppose that it executes one billion instructions per second

(设该计算机的运行速度为 10亿条基本指令/s )

- Input size:  $n = 3 \times 10^9$
- Insertion sort (插入排序): running time  $n^2$

$$t = s/v : \frac{3 \times 10^9 \times 3 \times 10^9 \text{ instruc}}{10^9 \text{ instruc/s}} = 9 \times 10^9 \text{ seconds} = \frac{9 \times 10^9}{60 \times 60 \times 24 \times 365} \text{ y} \approx 31.71 \text{ years}$$

# Identifying Genes in Human DNA (基因识别)

- Identifying all the 100,000 genes in human DNA
  - ◆ determining the sequences of the 3 billion( $10^9$ ) chemical base pairs that make up human DNA.

- Insertion sort (插入排序) : running time  $n^2$

$$\frac{3 \times 10^9 \times 3 \times 10^9 \text{ instruc}}{10^9 \text{ instruc/s}} = 9 \times 10^9 \text{ seconds} = \frac{9 \times 10^9}{60 \times 60 \times 24 \times 365} \text{ years} \approx 31.71 \text{ years}$$

- 
- Merge sort (归并排序) : running time  $n \lg n$

$$\frac{3 \times 10^9 \times \lg(3 \times 10^9) \text{ instruc}}{10^9 \text{ instruc/s}} = 3 \times \lg(3 \times 10^9) \text{ seconds} \approx 94.45 \text{ seconds}$$

- ✓ 两种排序算法都是可行的
- ✓ 但插入排序不适应于这个实际问题

# 查找问题

## 问题描述

从数据集合中找到满足某种条件的数据，返回两个结果：找不到返回空值，或找到返回搜索对象的位置。

## 查找算法

顺序查找，折半查找，二叉树查找等。

## 例子

从有序序列3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98中寻找元素70所在的位置。

## 简单的方法-顺序查找

- 序列中的元素依次和70比较，需比较8次。
- 对于 $n$ 个元素的序列，最坏情况下需要比较 $n$ 次。

# 顺序查找算法伪码

算法：顺序查找

```
1: function Seq_Search( $A[0\dots n - 1], K$ )
2:      $i \leftarrow 0$ 
3:     while  $A[i] \neq K$  do
4:          $i \leftarrow i + 1$ 
5:     end while
6:     if  $i < n$  then
7:         return  $i$  //首个值为 $K$ 的元素的位置
8:     else
9:         return  $-1$  //找不到返回-1
10:    end if
11: end function
```

# 折半查找

## 算法要点

对于有序的序列 $A$ ，待查找元素 $K$ 与序列最中间的元素 $A[m]$ 比较，如果相等算法结束；否则，如果 $K < A[m]$ ，在序列的前半部分查找，如果 $K > A[m]$ ，在序列的后半部分查找，如此重复。

例子：寻找 $K = 70$ 的元素

|   |   |    |    |    |    |    |    |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |
| 2 | 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |
| 3 | 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |

## 比较次数

上述例子寻找55需要4次比较；对于 $n$ 个元素的序列，最坏情况下的比较次数为 $\lfloor \log_2 n \rfloor + 1$ 。

# 折半查找算法伪码

## 算法：非递归的实现折半查找

```
1: function Binary_Search( $A[0\dots n - 1]$ ,  $K$ )
2:    $l \leftarrow 0$ ,  $r \leftarrow n - 1$ 
3:   while  $l \leq r$  do
4:      $m \leftarrow \lfloor (l + r)/2 \rfloor$ 
5:     if  $K = A[m]$  then
6:       return  $m$           //值为 $K$ 的元素的位置
7:     else if  $K < A[m]$  then
8:        $r \leftarrow m - 1$ 
9:     else
10:       $l \leftarrow m + 1$ 
11:    end if
12:  end while
13:  return  $-1$           //找不到返回-1
14: end function
```

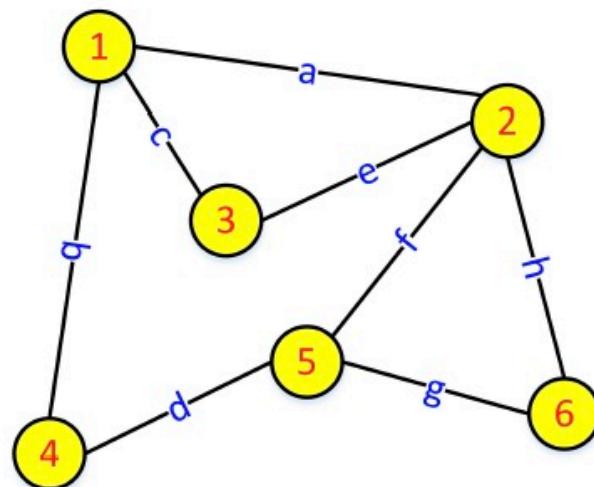
# 算法设计与分析的重要问题类型

- **排序问题**
- **查找问题**
- **图问题:** 主要涉及图的遍历和图的拓扑排序的内容
- **组合问题:** 要求从离散的空间中寻找一个对象,使之满足特定的标准(如满足某种最优化性质)
- **数值问题:** 涉及连续性的数学问题,解空间无穷大,近似解
- **几何问题:** 处理点、线、面这些对象的一些问题

# 图问题

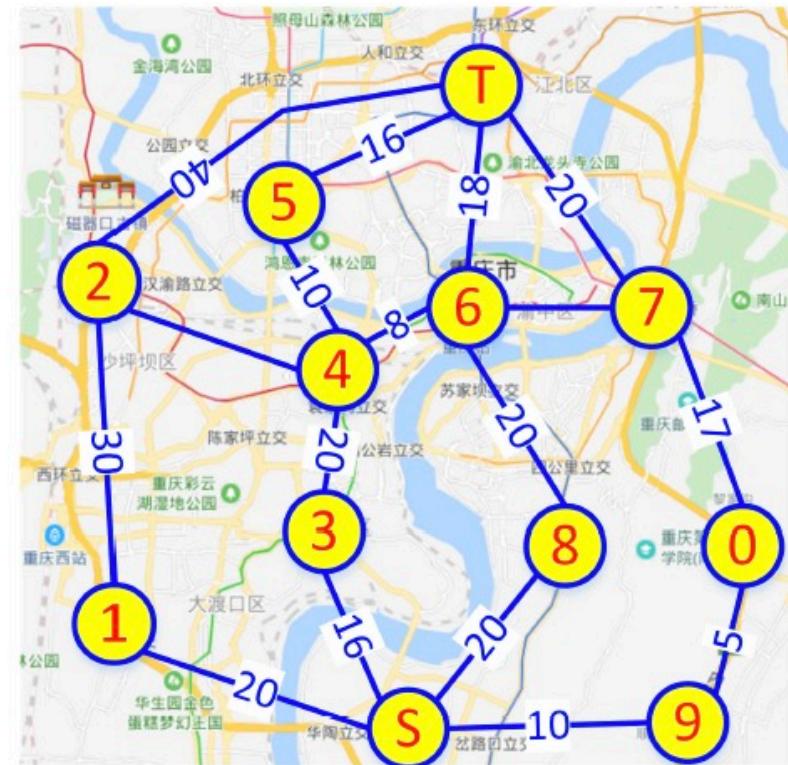
## 图的定义

图 (Graph) 是有一些称为顶点的点构成的集合，其中某些顶点由一些称为边的线段相连。



## 最短线路问题

两个地点之间的最佳线路是哪条？



# 组合问题

## 问题描述

寻找一些组合对象（例如一个排列，一个组合或者一个子集），这些对象能够满足特定的条件并具有我们想要的特性，如价值最大化或者成本最小化。图问题中的最短路径、旅行商和图着色问题都是组合问题。

## 组合问题是计算机领域中最难的问题

- 随着问题规模的增大，组合对象的数量增长极快。
- 还没有一种已知的算法能在可接受的时间内，精确地解决绝大多数这类问题。
- 有些组合问题有高效的算法求解（如最短路径算法），但是那仅仅是一些幸运的例外。

# 数值问题

## 典型的问题

解方程及方程组，计算定积分，求函数的值，矩阵乘法计算，矩阵链乘问题，最小公倍数，最大公约数，正整数相乘等。

## 问题难点

- 对于数值计算中的大多数问题，都只能近似求解；
- 数值问题要操作实数，而实数在计算机内部只能近似表示；
- 对近似数的大量算术操作可能会将大量的舍入误差叠加起来，导致一个看似可靠的算法输出严重歪曲的结果。

# 算法设计的主要策略

- 蛮力算法(穷举算法): 直接对问题进行求解
- 分治算法
- 贪心算法
- 动态规划算法
- 回溯算法
- 分支限界算法
- 随机化算法(概率算法)
- 线性规划算法
- NP完全性理论与近似算法
- 计算智能

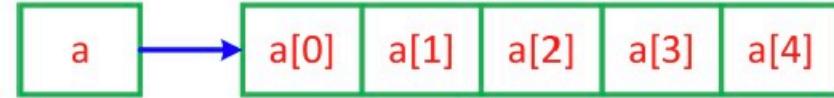
# 基本数据结构

- 列表
- 图
- 树
- 集合与字典
- 抽象数据类型

# 列表

## 数组

连续存储，访问快。



## 链表

便于插入和删除，内存占用小。



## 双向链表

便于寻找父节点。



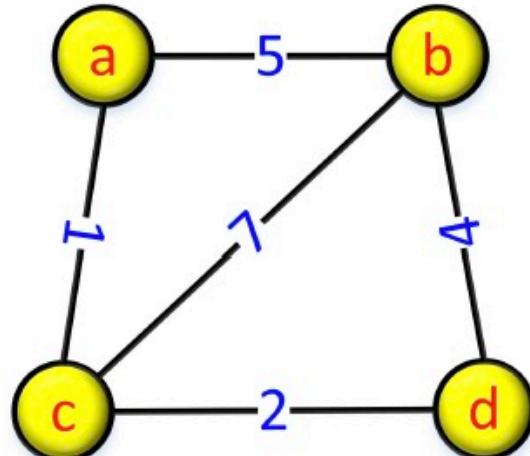
## 栈和队列

- 栈：先进后出，添加元素称为进栈，删除元素称为出栈；
- 队列：先进先出，添加元素称为入队，删除元素称为出队；

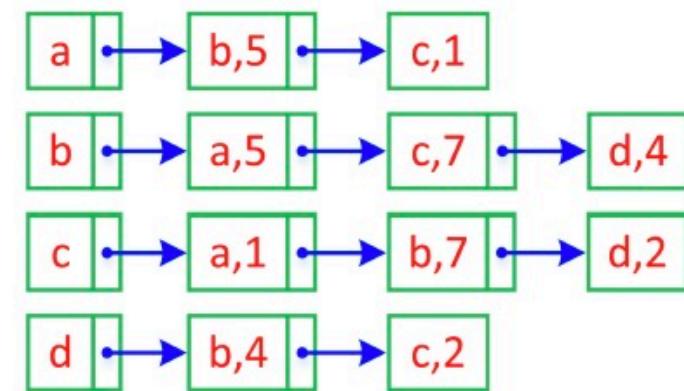
# 图

## 图的表示

- 图  $G = \langle V, E \rangle$  有两个集合来定义， $V$  表示顶点， $E$  表示边；
- 主要的图类型包括：无向图，有向图，加权图（边具有权值）；
- 图常常采用邻接矩阵或邻接链表表示；



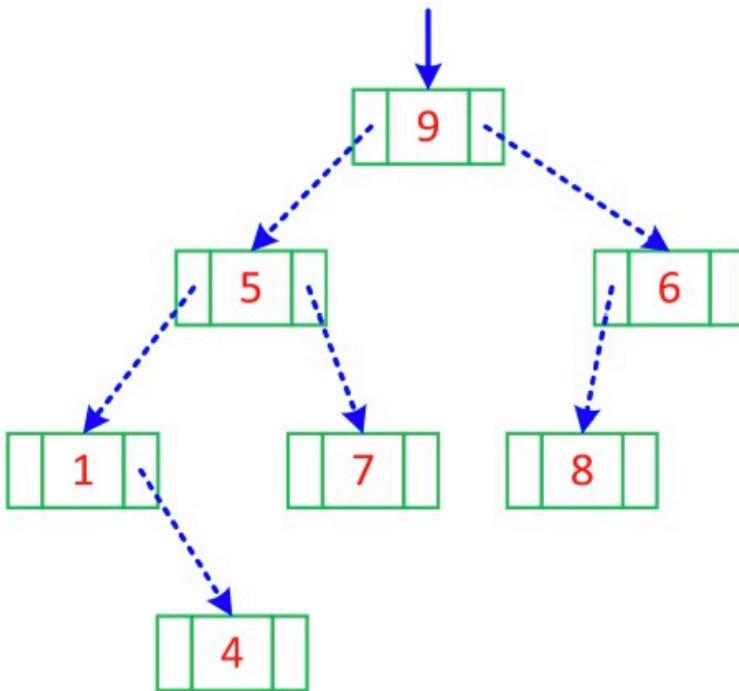
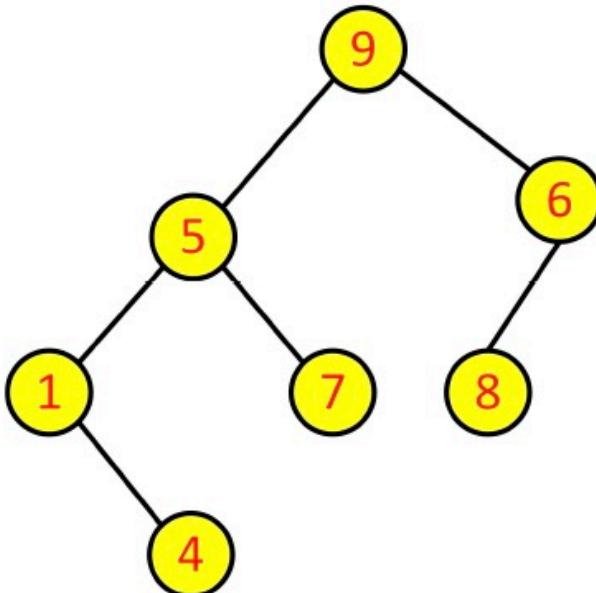
|   | a   | b   | c   | d   |
|---|-----|-----|-----|-----|
| a | Inf | 5   | 1   | Inf |
| b | 5   | Inf | 7   | 4   |
| c | 1   | 7   | Inf | 2   |
| d | Inf | 4   | 2   | Inf |



# 树

## 二叉树

二叉树的每个顶点的子女节点不超过两个，并且是有序的（分别为左子女和右子女节点）。



# 集合与字典

## 集合

集合是互不相同元素构成的无序集合，可以用位向量或者线性列表实现。最重要的集合操作有：检查元素是不是属于集合，集合求并集和交集。

## 字典

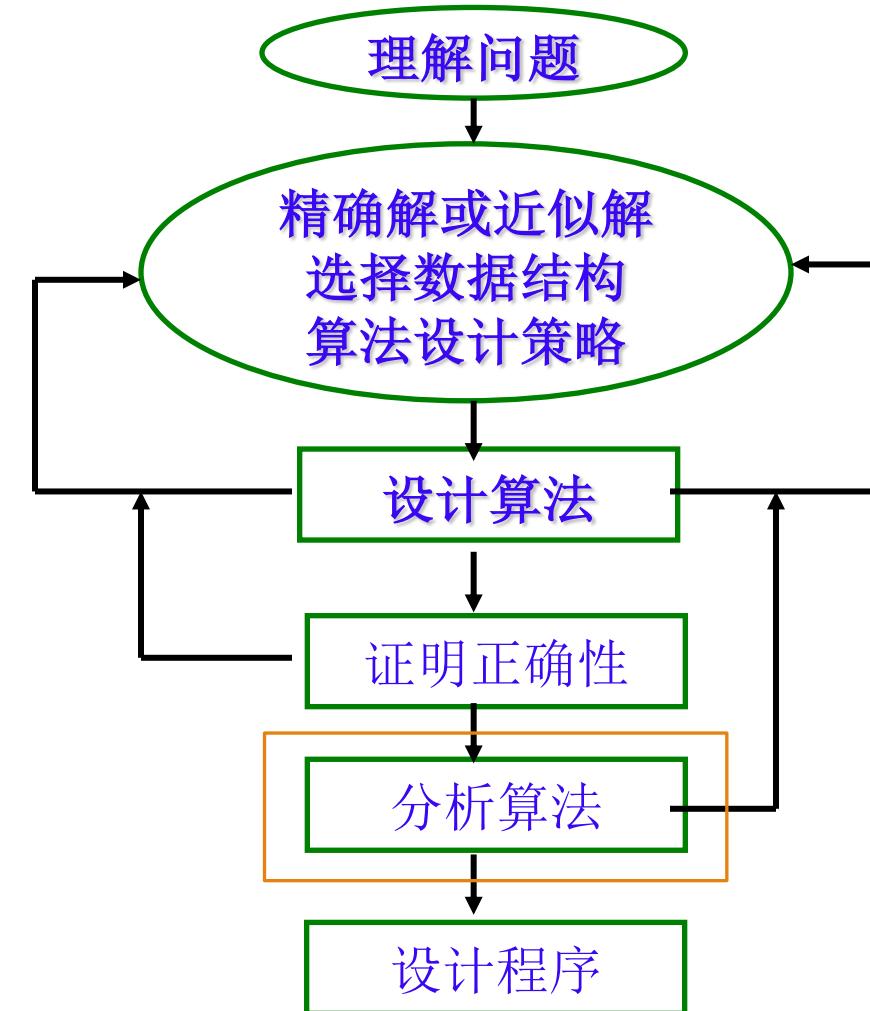
字典是以(key,value)的形式储存的数据结构，可方便地进行元素查找，及求并集和交集的操作。可以用数组，散列法和平衡查找树实现。

## 抽象数据类型

抽象数据类型是由一个表示数据项的抽象对象集合和一系列对这些对象的操作构成。在面向对象的编程语言（C++，Java）中，可用类实现。

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





# 算法分析

---

衡量算法的标准:

- 正确性
- 效率
  - 时间效率 (Time)
  - 空间效率 (Space)
- 简单性
- 一般性
- 优化性

# 算法效率分析

## 算法的效率分析包括两个方面

- 时间效率：讨论算法运行的有多快；
- 空间效率：关心算法占用的存储空间；

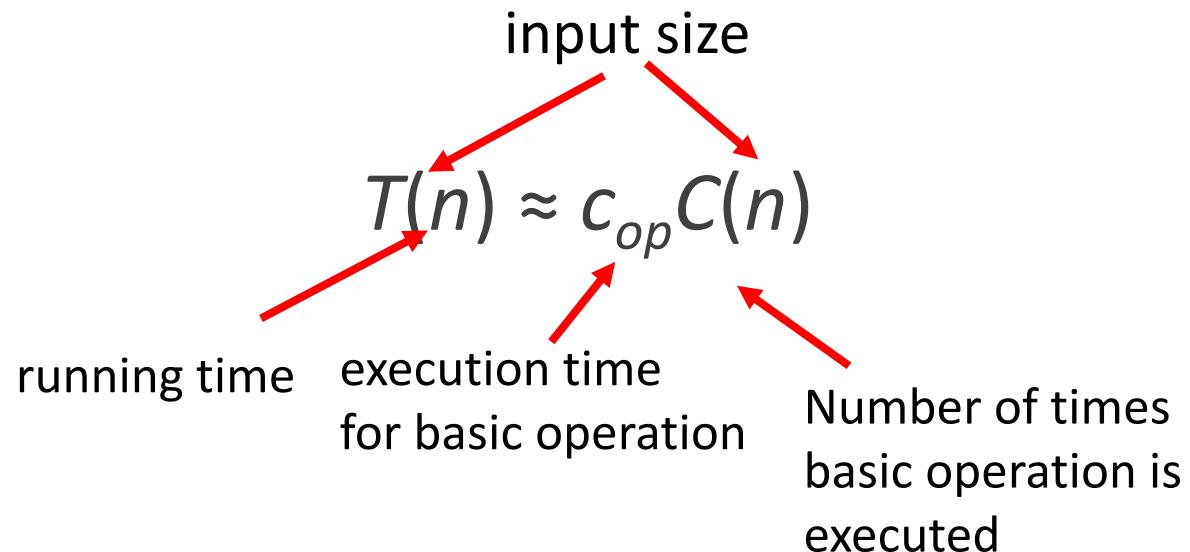
## 空间重要性降低

- 对于早期的计算机，时间和空间两种资源都是非常昂贵的。
- 随着电脑工艺的进步，计算机的存储容量已经提升了好几个数量级，降低了空间效率对算法性能的影响。
- 算法的时间效率没有得到相同程度的提高。

# 时间效率分析理论

时间效率分析是由以**输入规模**为函数的基本操作决定的

**基本操作**（加减乘除，除法最耗时，其次是乘法）的次数对算法的总运行时间贡献最大



# 典型问题的输入规模和基本操作

| <i>Problem</i>                            | <i>Input size measure</i>                               | <i>Basic operation</i>                  |
|---|---|---|
| Searching for key in a list of $n$ items  | Number of list's items, i.e. $n$                        | Key comparison                          |
| Multiplication of two matrices            | Matrix dimensions or total number of elements           | Multiplication of two numbers           |
| Checking primality of a given integer $n$ | $n$ 'size = number of digits (in binary representation) | Division                                |
| Typical graph problem                     | #vertices and/or edges                                  | Visiting a vertex or traversing an edge |

# 算法的三种效率

顺序查找算法从序列  $A[1, 2, \dots, n]$  中寻找元素 10

- 最好情况:  $A = [10, 2, 5, 6, 7, 8, 9]$ , 比较次数  $C_{best}(n) = 1$ ;
- 最坏情况:  $A = [5, 2, 5, 6, 7, 8, 10]$ , 比较次数  $C_{worst}(n) = n$ ;

## 算法三种效率

- 最差效率  $C_{worst}(n)$ : 输入规模为  $n$  时, 算法在**最坏**输入下的效率;
- 最优效率  $C_{best}(n)$ : 输入规模为  $n$  时, 算法在**最好**输入下的效率;
- 平均效率  $C_{avg}(n)$ : 输入规模为  $n$  时, 算法在**随机**输入下的效率;

**最差时间效率比平均时间效率计算简单**

**平均时间效率可能更符合实际**

**最好时间效率一般很少使用**

# 最坏和平均情况分析

- 一般情况下我们关注最坏情况下的运行时间 (即, 对于任意输入size  $n$ , 算法花费的最长时间).
  - 最坏情况是算法对于任意输入运行时间的上界 (upper bound)
  - 实际上, 最坏情况经常发生 (in searching a database)
  - 平均情况通常和最坏情况一样差 (例如, 我们随机选择  $n$  个数进行插入排序)
- 平均情况是一个算法的期望运行时间
  - 不是最坏和最好情况的平均
  - 是基于概率的分析
  - 算法的随机化分析



# 时间效率分析

---

- 假设赋值语句 (`max=a[0]`) 和条件语句 (`if`) 都分别需要占用一个时间单元
- 而循环语句使用3个时间单元：1个初始化，判断语句和增量语句

```
max=a[0];
for (i=1; i<n; i++)
{
    if (a[i]>max)
        max=a[i];
}
```



# 时间效率分析

---

要计算  $T(n)$ , 需要做:

- 1 time unit for the initialization of max.
- 1 time unit for the initialization of the for loop.
- 1 time unit for the initial for loop test (i.e.  $i < n$ ;
- 2 time unit for the body of the loop

```
max=a[0];
for (i=1; i<n; i++)
{
    if (a[i]>max)
        max=a[i];
}
```



# 时间效率分析

- 最坏情况 Worst case:

最坏情况下，每次迭代都要执行循环体内部的语句

- 如果  $a=\{1,2,3,4,5,6,7\}$ , 每次迭代需要做4步

1 for the loop test

1 for the increment

1 for the if

1 for the assignment.

- 所以最坏情况下时间效率为:  $4(n-1)$

```
max=a[0];
for (i=1; i<n; i++)
{
    if (a[i]>max)
        max=a[i];
}
```

# 时间效率分析

---

平均情况 Average case:

平均情况下，假设循环迭代过程中每两个元素会替代max

假设  $a=\{5,2,7,4,8,6,9\}$ , 那么每次迭代需要 3 个时间单元(2 for loop control and 1 for the if) 和  $(n-1)/2$  的 max 赋值.

所以时间效率为:  $3(n-1) + (n-1)/2 = 3.5(n-1)$ .

```
max=a[0];
for (i=1; i<n; i++)
{
    if (a[i]>max)
        max=a[i];
}
```



# 时间效率分析

---

## 最好情况 Best case:

- 最好情况下， $a[0]$  就是最大元素，所以 `max` 赋值语句将不被执行，
- 时间效率为  $3(n-1)$ .

```
max=a[0];
for (i=1; i<n; i++)
{
    if (a[i]>max)
        max=a[i];
}
```



# 时间效率分析

---

- $T_{worst}(n) = 4(n-1) + 3 = 4n - 4 + 3 = 4n - 1$
- $T_{avg}(n) = 3.5(n-1) + 3 = 3.5n - 3.5 + 3 = 3.5n - 0.5$
- $T_{best}(n) = 3(n-1) + 3 = 3n - 3 + 3 = 3n$

# 平均时间复杂度

## 数学定义

Let  $D_n$  be the set of inputs of size  $n$  for the problem under consideration.

Let  $I$  be an element of  $D_n$  and let  $p(I)$  be the probability that input  $I$  occurs.

Let  $t(I)$  be the number of basic operations performed by the algorithm of input  $I$ . Then its average behavior may be realistically defined as

$$A(n) = \sum_{I \in D_n} p(I) \bullet t(I)$$



# 顺序查找算法的平均时间效率分析

---

$$\begin{aligned}C_{avg}(n) &= p * [1 * \frac{1}{n} + 2 * \frac{1}{n} + \dots + n * \frac{1}{n}] + (1 - p) * n \\&= \frac{p(n + 1)}{2} + n(1 - p)\end{aligned}$$

注： $p$ 为查找成功的概率， $p = 1$ 表示查找元素一定在序列中， $p = 0$ 表示会查找不成功。



# 算法复杂性分析

最坏情况下的时间复杂性：

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

最好情况下的时间复杂性：

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

平均情况下的时间复杂性：

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

其中  $D_N$  是规模为  $N$  的合法输入的集合；  $I^*$  是  $D_N$  中使  $T(N, I^*)$  达到  $T_{\max}(N)$  的合法输入；  $\tilde{I}$  是中使  $T(N, \tilde{I})$  达到  $T_{\min}(N)$  的合法输入； 而  $P(I)$  是在算法的应用中出现输入  $I$  的概率。

# 算法效率度量——增长次数

## 思考

考虑两个算法，计算复杂度分别为 $T_1(n) = n$ ,  $T_2(n) = n!$ , 当 $n = 1$ 和 $2$ 的时候，二者的运行时间相同，它们的计算效率是不是一样的？当输入规模变大， $T_1(10) = 10$ ,  $T_2(10) \approx 3.6 * 10^6$ , 是不是意味着算法1比算法2快360万倍？

## 结论

- 小规模的输入不足以将高效的算法和低效的算法区分出来，必须考虑较大规模的输入；
- 不仅需要知道算法需要多少时间运行完毕，还需要知道运行时间随着输入规模的增长而增加的幅度；

# 近似运行时间分析

- $T(n)$  忽略了每条语句由不同编译器/解释器，及硬件环境带来的时间差异
- 所以不能说某个程序需要花费多少秒的运行时间，除非我们知道程序运行的硬件及编译解释环境
- 完全精确的运行时间估计不太可能的

# 函数增长率 (Rate of growth, order of growth)

---

- How much faster will algorithm run on computer that is twice as fast?  
(机器快2倍)
- How much longer does it take to solve problem of double input size?  
(算法输入规模大2倍)

# 与机器无关的运行时间

- 演近分析思想:
  - 忽略与机器相关的常数项 Ignore machine-dependent constants.
  - 看当  $n \rightarrow \infty$  时  $T(n)$  的增长
  - 只考虑效率函数公式中的主项 (例如最高阶项,  $an^2$  of  $an^2+bn+c$ )
  - 忽略主项的常数系数 (?)
  - 如果一个算法最坏情况的渐近增长率比另一个算法的低, 那么这个算法更高效

# 增长次数描述：渐进符号

## 上界 $O$ 的定义

如果存在大于0的常数 $c$ 和非负整数 $n_0$ ，使得 $\forall n > n_0$ ,  $t(n) \leq cg(n)$ ，我们称函数 $t(n)$ 包含在 $O(g(n))$ 中，记作 $t(n) \in O(g(n))$ 。

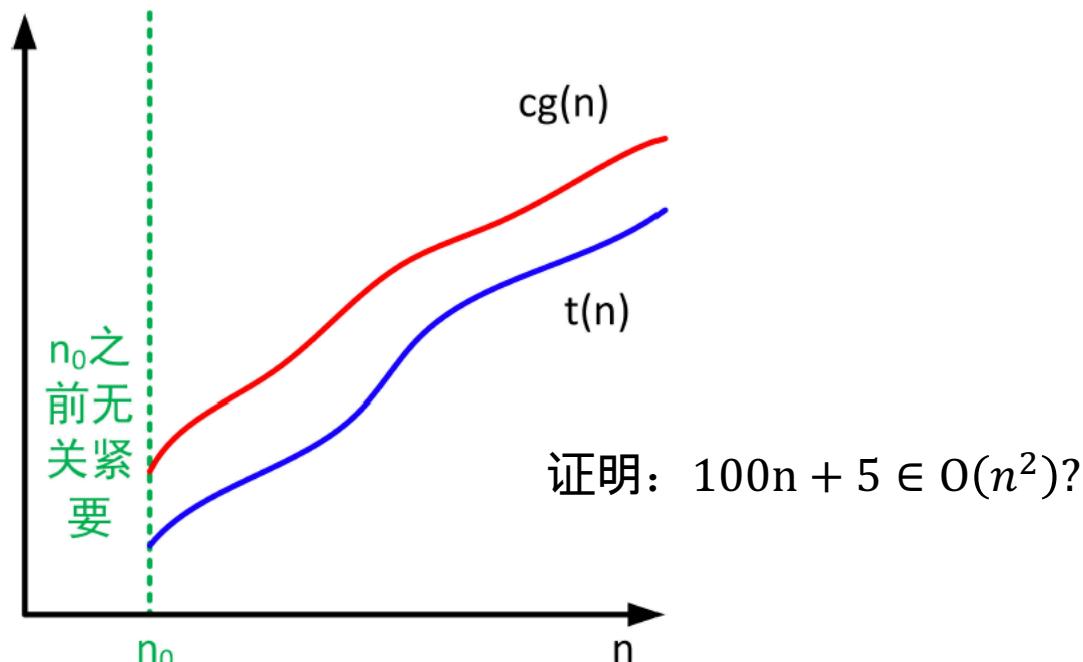
## 例子

$$n \in O(n), n \in O(n^2), n \in O(n^3)$$

$$100n + 5 \in O(n^2)$$

$$0.5n(n+1) \in O(n^2)$$

$$n^2 \notin O(n), n^3 \notin O(n^2)$$



# 增长次数描述：渐进符号

## 下界Ω的定义

如果存在大于0的常数 $c$ 和非负整数 $n_0$ ，使得 $\forall n > n_0$ ,  $t(n) \geq cg(n)$ ，我们称函数 $t(n)$ 包含在 $\Omega(g(n))$ 中，记作 $t(n) \in \Omega(g(n))$ 。

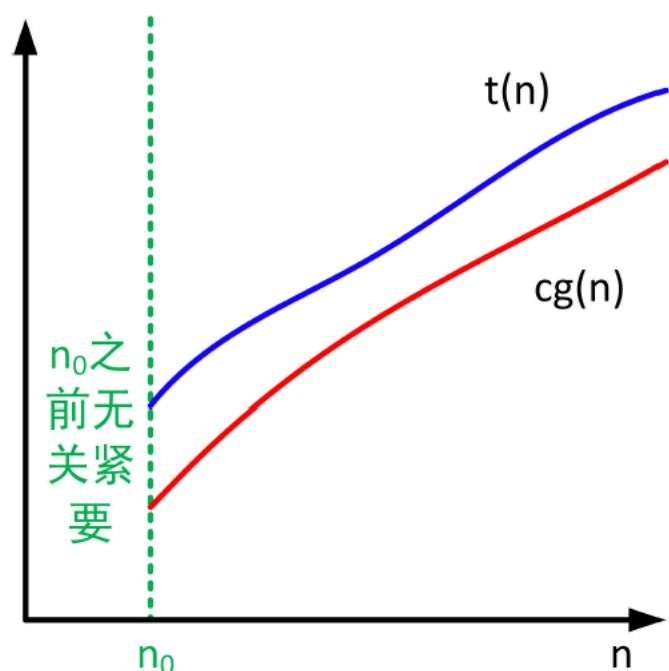
## 例子

$$n^2 \in \Omega(n)$$

$$0.5n(n - 1) \in \Omega(n^2)$$

$$n^3 \in \Omega(n), n^3 \in \Omega(n^2)$$

$$100n + 5 \notin \Omega(n^2)$$



# 增长次数描述：渐进符号

## $\Theta$ 的定义

若存在大于0的常数 $c_1, c_2$ 和非负整数 $n_0$ ,  $\forall n > n_0, c_1g(n) \leq t(n) \leq c_2g(n)$ , 我们称函数 $t(n)$ 包含在 $\Theta(g(n))$ 中, 记作 $t(n) \in \Theta(g(n))$ 。

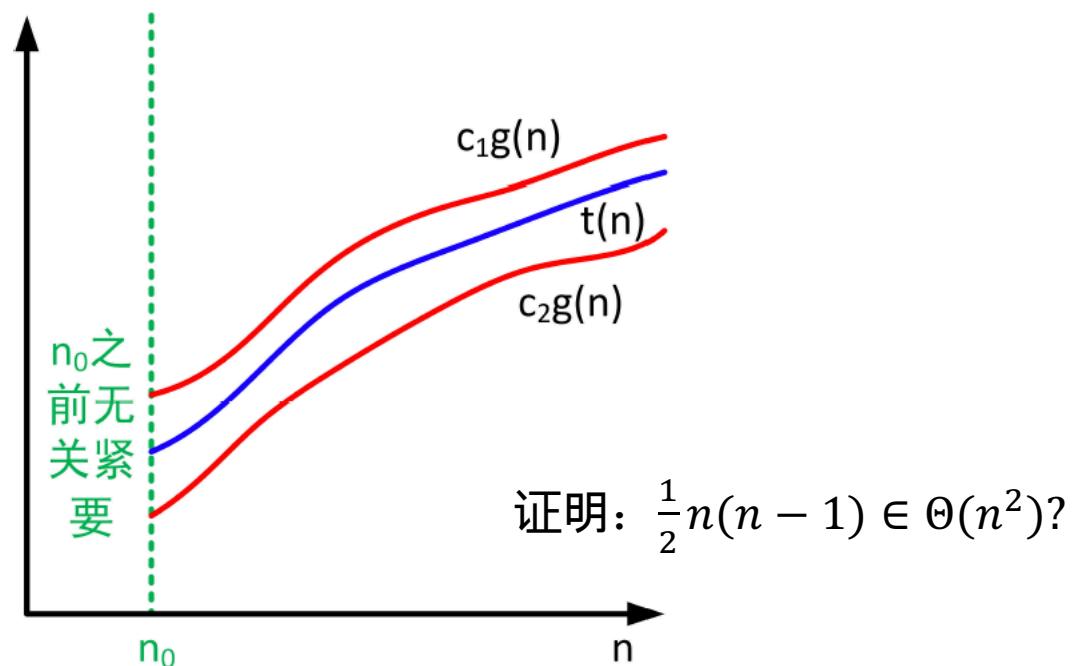
## 例子

$$100n^2 \in \Theta(n^2)$$

$$0.5n(n - 1) \in \Theta(n^2)$$

$$0.5n(n - 1) \notin \Theta(n^3)$$

$$0.5n(n - 1) \notin \Theta(n)$$





# 渐进符号特性

## 定理

如果  $t_1(n) \in O(g_1(n))$ , 并且  $t_2(n) \in O(g_2(n))$ , 则

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

对于  $\Omega$  和  $\Theta$  符号, 类似的断言也成立。

## 例子：检查数组中是否含有相同元素

- 算法首先对数组进行排序, 然后比较相邻元素是否相等;
- 排序算法采用选择排序, 比较次数为  $C_n^2 \in O(n^2)$ ;
- 比较相邻元素步骤的比较次数为  $n - 1 \in O(n)$ ;
- 算法的整体效率为  $O(\max\{n^2, n\}) \in O(n^2)$
- 结论: 算法的整体效率由较大增长次数 (效率较差) 的部分决定。

# 利用极限比较增长次数

## 定理

设 $f$ 和 $g$ 是定义域为自然数集合 $N$ 上的非负函数，有：

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & : t(n) \in O(g(n)) \\ c > 0 & : t(n) \in O(g(n)); t(n) \in \Omega(g(n)); t(n) \in \Theta(g(n)) \\ \infty & : t(n) \in \Omega(g(n)) \end{cases}$$

## 例子

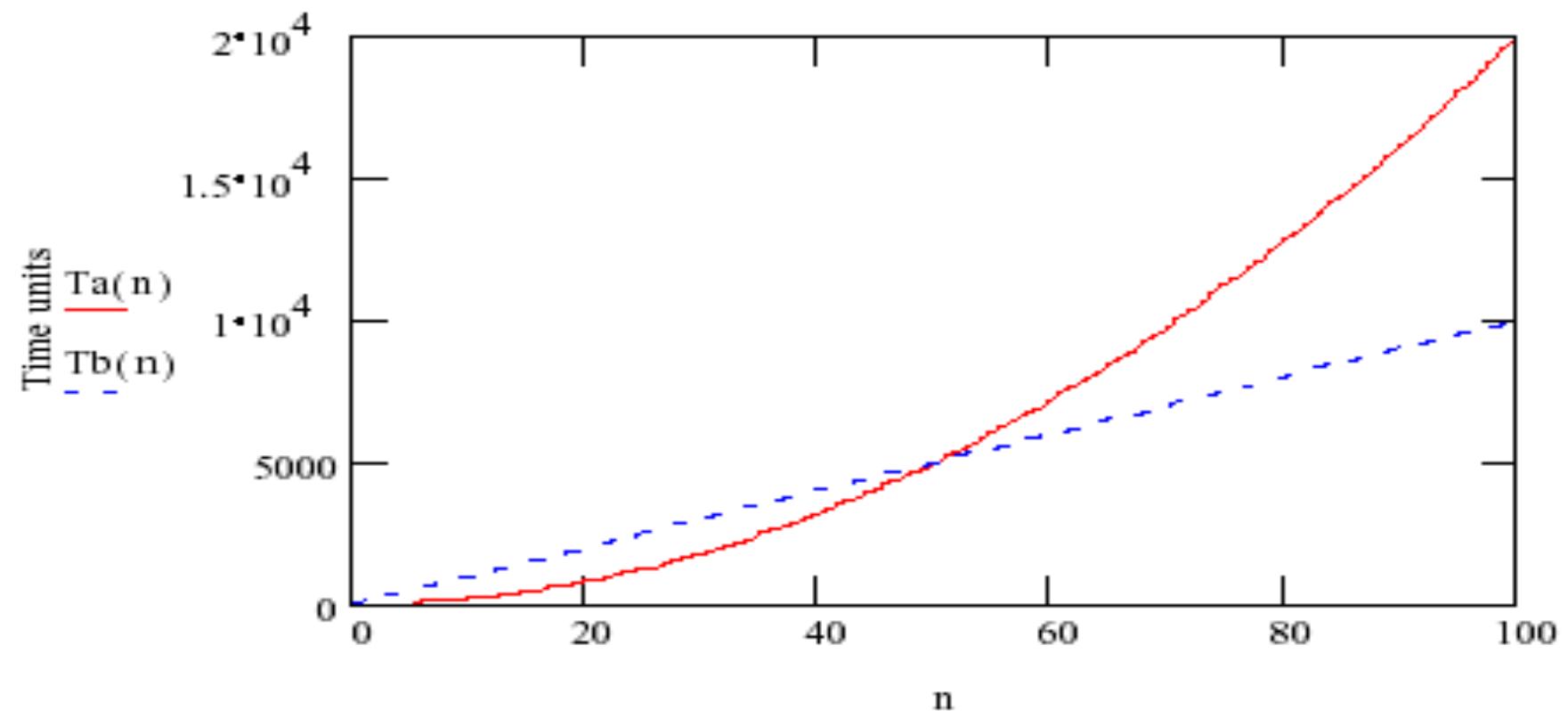
$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{\log_2 n'}{\sqrt{n'}} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$
$$\log_2 n \in O(\sqrt{n})$$

# 基本效率类型

| 类型         | 名称   | 注释                                     |
|------------|------|--|
| 1          | 常数   | 为数很少的效率最高的算法，难以举例。                     |
| $\log n$   | 对数   | 算法的每一次循环都会消去问题规模的一个常数因子，如有序序列中的折半查找算法。 |
| $n$        | 线性   | 扫描规模为 $n$ 的列表，如顺序查找算法。                 |
| $n \log n$ | 线性对数 | 许多分治算法（合并排序，快速排序）的平均效率属于这个类型。          |
| $n^2$      | 平方   | 一般来说，包含两重嵌套循环算法的典型效率，选择排序和冒泡排序属于这一类型。  |
| $n^3$      | 立方   | 一般来说，包含三重嵌套循环算法的典型效率。                  |
| $2^n$      | 指数   | 求 $n$ 个元素集合的所有子集的算法，如蛮力查找示解背包问题和最近对问题。 |
| $n!$       | 阶乘   | 求 $n$ 个元素的完全排列算法，如穷举查找示解分配问题，旅行商问题。    |

# $2n^2$ and $100n$

---



# 重要函数特征

随输入规模的增长情况（近似）

| $n$    | $\log_2 n$ | $n$    | $n \log_2 n$ | $n^2$     | $n^3$     | $2^n$           | $n!$             |
|--------|------------|--------|--------------|-----------|-----------|-----------------|------------------|
| 10     | 3.3        | 10     | $3.3 * 10$   | $10^2$    | $10^3$    | $10^3$          | $3.6 * 10^6$     |
| $10^2$ | 6.6        | $10^2$ | $6.6 * 10^2$ | $10^4$    | $10^6$    | $1.3 * 10^{30}$ | $9.3 * 10^{157}$ |
| $10^3$ | 10         | $10^3$ | $1.0 * 10^4$ | $10^6$    | $10^9$    |                 |                  |
| $10^4$ | 13         | $10^4$ | $1.3 * 10^5$ | $10^8$    | $10^{12}$ |                 |                  |
| $10^5$ | 17         | $10^5$ | $1.7 * 10^6$ | $10^{10}$ | $10^{15}$ |                 |                  |
| $10^6$ | 20         | $10^6$ | $2.0 * 10^7$ | $10^{12}$ | $10^{18}$ |                 |                  |

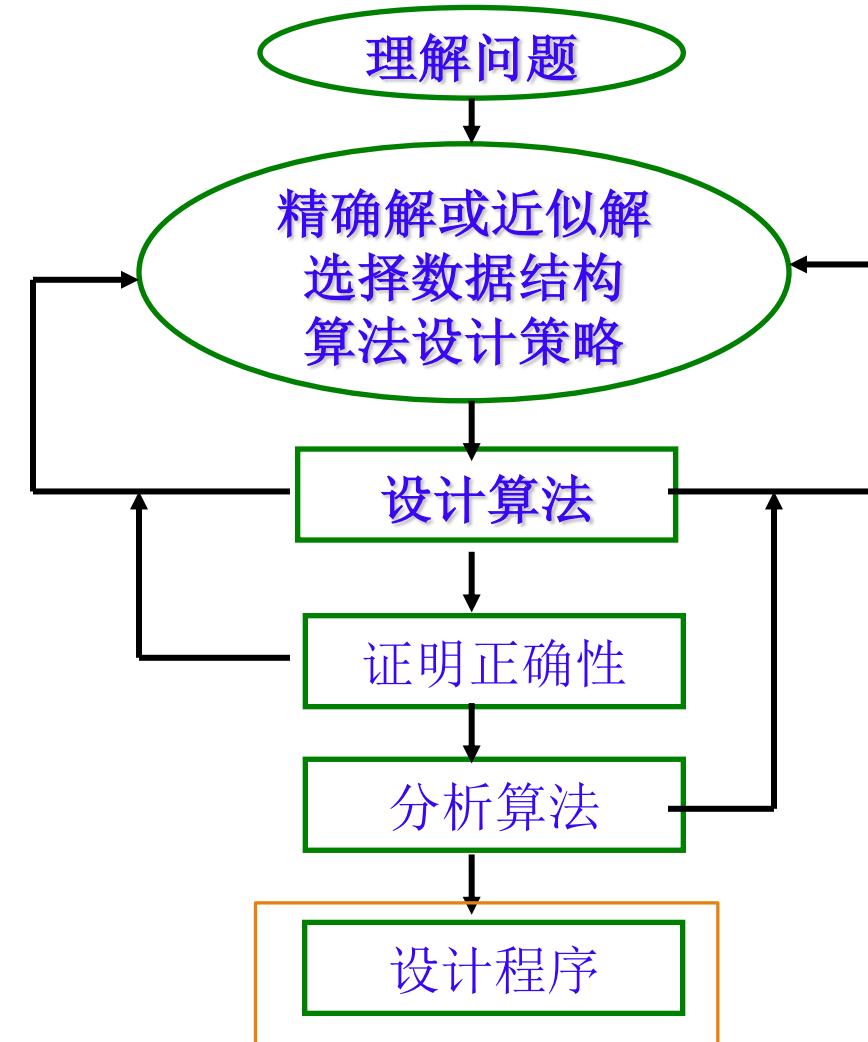
注：当 $n$ 增长为 $2n$ 时，增长分别是： $\log_2 n$ 为1， $n$ 为2倍， $n \log_2 n$ 为2倍多， $n^2$ 为4倍...

## 注意

对数函数的操作次数依赖于对数的底，由于 $\log_a n = \log_b a * \log_b n$ ，因此底不相同的两个对数函数只差一个乘法常量，当我们关心增长次数的时候，忽略对数的底，简单写成 $\log n$ 。

# 问题求解 (Problem Solving)

- 理解问题是解决问题的关键；
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律；
- 发明算法是要一个非常有创造性和非常值得付出的过程；





# 设计程序实现算法

---

- 选择一种语言编程实现
- 对算法进行测试

# 小结

- 算法是一些列解决问题的明确指令，对于符合一定规范的输入，能够在有限时间内获得要求的输出；
- 理解问题是利用算法解决问题的首要条件，算法设计是一个不断重复的过程；
- 常见的重要问题类型有：排序，查找，字符串处理，图问题，组合问题，几何问题和数值问题；
- 算法分析主要包括时间复杂度和空间复杂度，而时间复杂度是算法课程的核心内容；
- 算法运行时间的度量单位为最影响算法性能的基本操作；
- 增长次数是衡量算法好坏的最关键的指标；
- 算法的效率包括最优效率，最差效率和平均效率；

# 作业 1

- 阅读
  - 1.1–1.4 2.1–2.3
- 习题
  - 1.1: 3b, 4
  - 1.2: 4, 5
  - 1.4: 3, 4, 9
  - 2.3: 4, 5, 6
- 思考: 习题 1.3 4, 5