数据结构与算法

第 1 章 概论

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

国家精品课“数据结构与算法”

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6, “十一五”国家级规划教材

算法

算法

问题 —— 算法 —— 程序

- **问题（problem）** 一个函数
  - 从输入到输出的一种映射
- **算法（algorithm）** 一种方法
  - 对特定问题求解过程的描述，是指令的有限序列
- **程序（program）** 一种实现
  - 算法在计算机中以某种程序设计语言的一种实现

算法

- 对特定问题求解过程的描述，是指令的有限序列；也即，为解决某一特定问题而采取的具体有限的操作步骤
  - 算法的描述可以采用多种方式：

    语言（伪码）、图形、表格等

- 程序是算法的一种实现，计算机按照程序逐步执行算法，实现对问题的求解

算法的特性

- **通用性**
  - 对于符合输入类型的任意输入，都能根据算法进行问题求解，并保证计算结果的正确性
- **有效性**
  - 组成算法的每一条指令均须能够被（人或机器）确切执行，且指令的类型应该明确规定，其结果应具有确定的数据类型，是能够预期的
- **确定性**
  - 算法描述中的下一步应执行的步骤必须明确
- **有穷性**
  - 算法的执行须在有限步内结束，即不能含有死循环

算法设计分类

- **算法设计与算法分析是计算机科学的核心问题**
- **常用的设计方法**
  - 穷举法　(百钱买百鸡)
  - 贪心法　(Huffman树)
  - 递归法, 分治法(二分法检索)
  - 回溯法(八皇后)
  - 动态规划法(最佳二叉排序树)
  - 宽度优先和深度优先搜索
  - α-β裁剪和分枝界限法
  - 并行算法

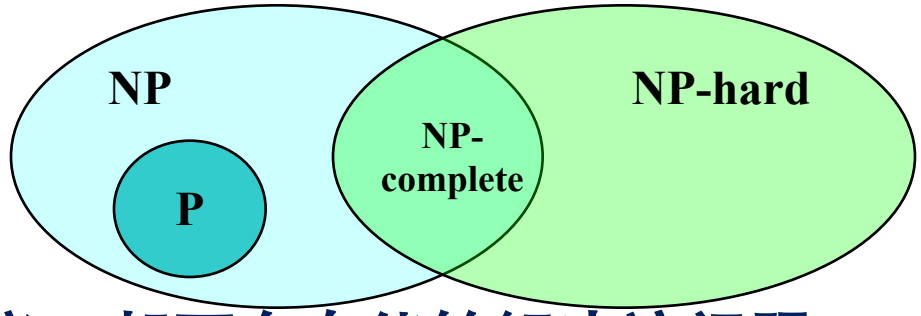
算法设计分类

- **穷举法**（百钱买百鸡，顺序找 K 值） — 万能，低效
  - 避免穷举测试
- **回溯**（迷宫、八皇后）、**搜索**（DFS, BFS）
  — 跳过无解分支
  - 最优化问题的通法
- **递归分治**（二分检索、快速排序、分治排序）
  — 自顶向下，问题化解
  - 子结构不重复
  - 分、治、合

算法设计分类

- **动态规划（Floyd算法）**
  — 自底向上，利用中间结果，迅速构造
  - 最优子结构、重复子结构、无后效性
  - 搜索中分支定界的特例
  - 空间换时间
- **贪心法（Huffman最优编码，Dijkstra最短路径）**
  — 动态规划的特例
  - 最优子结构——最优解
  - 若非则只是快速得到较优解

计算复杂性理论



- **不可解问题：虽能被准确定义，却不存在能够解决该问题的算法**
  - 停机问题
- **难解问题：求解算法均无法在多项式时间 n^k 数量级内解决（其中 k 是任意正整数）**
  - 最优巡游路径问题
  - 判定命题逻辑公式是否恒真
  - ……
- **组合爆炸型的难解问题**
  - 随着问题的规模 n 的增大，算法的时间开销不能约束在 n 的 k 阶多项式数量范围内（常数k不依赖于 n ）
  - 八皇后、Hanio塔

算法的效率及其度量

- **解决同一个问题存在着多种算法，如何评估算法的好坏？或据此设计出更好的算法？**
  - 运行该算法所需要的计算机资源的多寡，所需越大复杂性越高
  - 最重要的资源：时间（处理器）和空间（存储器）
- **评价一个算法优劣的重要依据是看执行该算法的程序需要占用多少机器资源：**
  - 程序所用算法运行时所要花费的时间代价
  - 程序中使用的数据结构占有的空间代价

算法的效率及其度量

- 需明确
  - 如何表达
  - 怎样计算

一个算法的复杂性

算法复杂性表示

- **时间复杂性不能用诸如微秒、纳秒这样的真实时间单位**
  - 相同算法
    - 运行在**不同的机器**： Cray机 vs. PC
    - 运行在**同样的机器**： 使用程序语言不同， C vs LISP
  - 效率差异明显的算法运行在不同的机器上
    - 一个运行在Cray机上的效率极差的算法也许比一个运行在PC上的效率很高的算法花费更少的时间
  - 两个不同的算法也许在输入规模为100 时表现不相上下，而在输入规模扩大10倍后却表现迥异

算法复杂性表示

- 算法分析感兴趣的并非**具体的资源占用量**，而是与**具体平台无关、具体输入无关、且随输入规模增长可预测的趋势**
  - **与问题规模的关系**，e.g., 时间效率可用一定“规模”(size)”的数据作为输入时程序运行所需的“基本操作(basic operation)” 数目，e.g.,
    - 若数据规模n和运行时间 t 间存在一个线性关系：t = cn，则数据规模增加 5 倍后，运行时间相应增加同样的倍数，即 n’ = 5n ➔ t’ = 5t
    - 若 t = $\log_2 n$，则 n 的规模加倍意味着运行时间随之增加一个单位，i.e.，n' = 2n ➔ t' = $\log_2 (2n) = \log_2 n + 1 = t + 1$
  - **完成一个“基本操作”所需时间与具体的被操作数无关**

算法的渐近分析

- Asymptotic analysis，**简称算法分析**
  - 由于算法的复杂性与其所求解问题的规模直接有关，因此通常将问题规模 n 作为一个参照量，求算法的时空开销和 n 之间的关系
  - 一般情况下，这种函数关系都相当复杂，故计算时只考虑那些显著影响函数量级的部分，即结果为原函数的一个近似值
- 对资源开销的一种**不精确估计**，提供对于算法资源开销进行评估的简化模型

算法的渐近分析

$$f(n) = n^2 + 100n + \log_{10}n + 1000$$

- 数据规模 *n* 逐步增大时，资源开销函数 *f(n)* 的增长趋势
- 当 *n* 增大到一定值以后，计算公式中对计算结果影响最大的就是幂次最高的项
  - 其他的常数项和低幂次项都可以忽略

算法渐近分析：大O表式法

- 假设f 和g为从自然数到非负实数集的两个函数

  定义1：如果存在正数*c*和n_0，使得对任意的$n \geq n_0$，都有

  $$f(n) \leq cg(n),$$

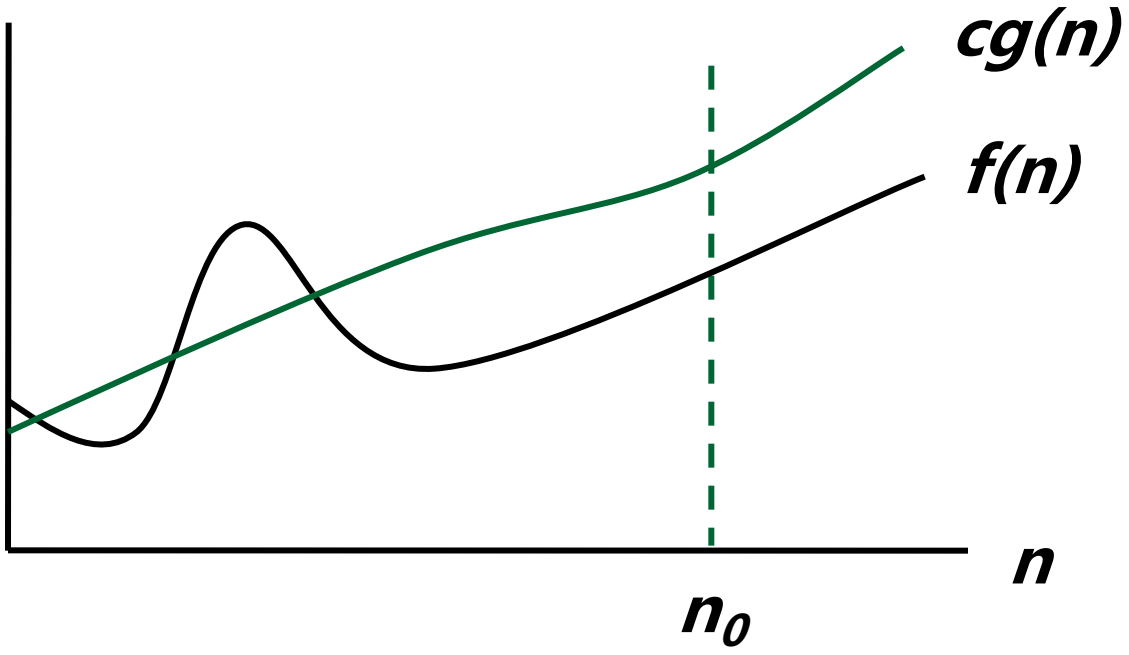
  则称f(n)在集合O(g(n))中，简称f(n)是O(g(n))的，一般记为 f(n) = O(g(n))

  - 函数g(n) 是函数 f(n) 取值的上限（*upper bound*），或说函数 f 的增长最终至多趋同于函数 g 的增长
  - 一种表达函数增长率上限的方法
  - 一个函数增长率的上限可能不止一个。大O表示法给出了所有上限中最“紧”（也即最小）的那个上限

算法渐近分析：大O表式法

$f(n) = O(g(n))$, iff

$\exists\, c, n_0 > 0$ s.t. $\forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$



f(n) is *eventually* upper-bounded by *g(n)*

大O表示法的单位时间

- **简单布尔 或 算术运算**
- **简单I/O**
  - 指函数的输入/输出
    - e.g.，从数组读数据等操作
  - 不包括键盘、文件等I/O
- **函数返回**

大O表式法的性质

- **若符号 *a* 是不依赖于 *n* 的任意常数**

1. 若 f(n) = O(g(n))，g(n) = O(h(n))，则f(n)=O(h(n))
2. 若 f(n) = O(h(n))，g(n) = O(h(n))，则

   f(n) + g(n) = O(h(n))
3. 函数an^k 是 $O(n^k)$的
4. 对任何正数 j 而言，函数n^k 是$O(n^{k+j})$的
5. 对于任何正数a和b，且b≠1，函数$\log_a n = O(\log_b n)$，即任何对数函数，无论底数为何，都具有相同的增长率
   - 对任何正数 a≠1，都有 $\log_a n = O(\lg n)$，其中 $\lg n = \log_2 n$

大O表示法的运算规则

- **加法规则：** $f_1(n)+f_2(n) = O(\max(f_1(n), f_2(n)))$
  - 顺序结构，if 结构，switch结构
- **乘法规则：** $f_1(n)\cdot f_2(n) = O(f_1(n)\cdot f_2(n))$
  - for, while, do-while结构

```
for (i=0; i<n; i++)
    for (j=i; j<n; j++)
        k++;
```

$n-i$

?

$$\sum_{i=0}^{n-1}(n-i) = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

算法渐近分析： Ω表式法

- **定义2**： 若存在正数c 和n_0，使对所有的$n \geq n_0$，都有 $f(n) \geq cg(n)$，

  则称f(n)在集合Ω (g(n)) 中，或简称f(n) 是 Ω (g(n))的，记为 f(n) = Ω (g(n))

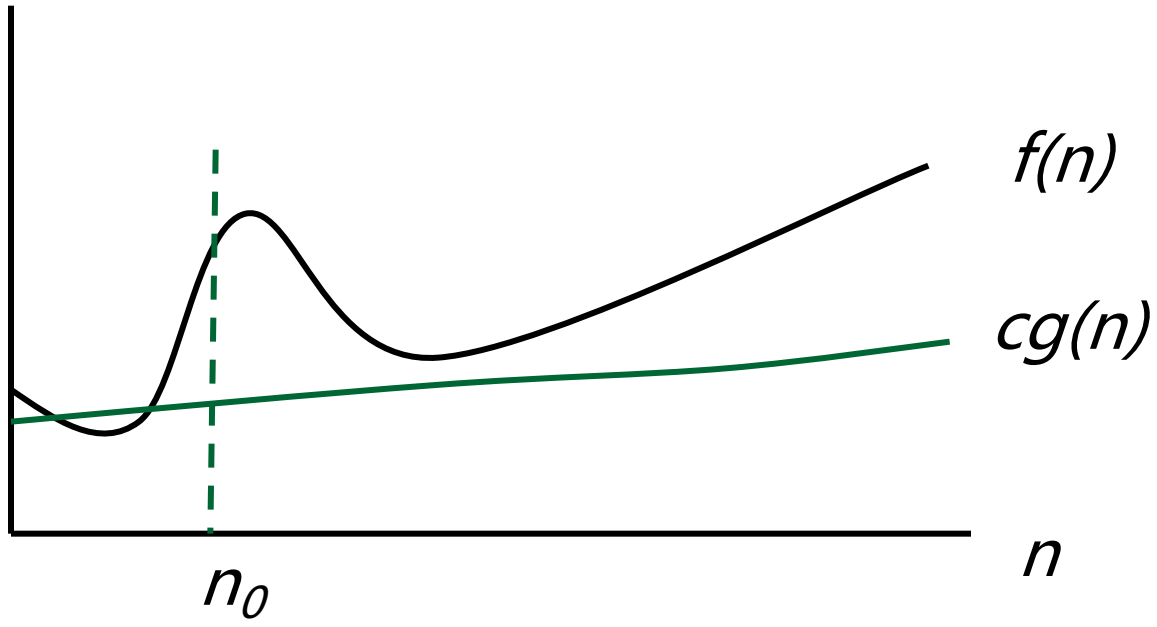
  - cg(n)是函数f(n)取值的下限（*lower bound*），或说，函数f的增长最终至少趋同于函数g的增长
  - 一种表达函数增长率下限的方法
  - 正如大O表示法，Ω表示法也是在函数增值率的所有下限中那个最“紧”（即最大）的下限

Ω表式法

$f(n) = \Omega(g(n))$, iff

$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n0 , 0 \leq cg(n) \leq f(n)$

与大O表示法的唯一区别在于 不等式的方向



f(n) is eventually lower-bounded by g(n)

算法渐近分析： Θ表式法

- 当上、下限相同时则可用 Θ 表示法
- **定义3**:

  如果一个函数既在集合O (g(n))中又在集合Ω (g(n))中，则称其为Θ (g(n))。 记为f(n) = Θ(g(n))

  存在正常数c_1，c_2，以及正整数n_0，使得对于任意的正整数$n > n_0$，有下列两不等式同时成立

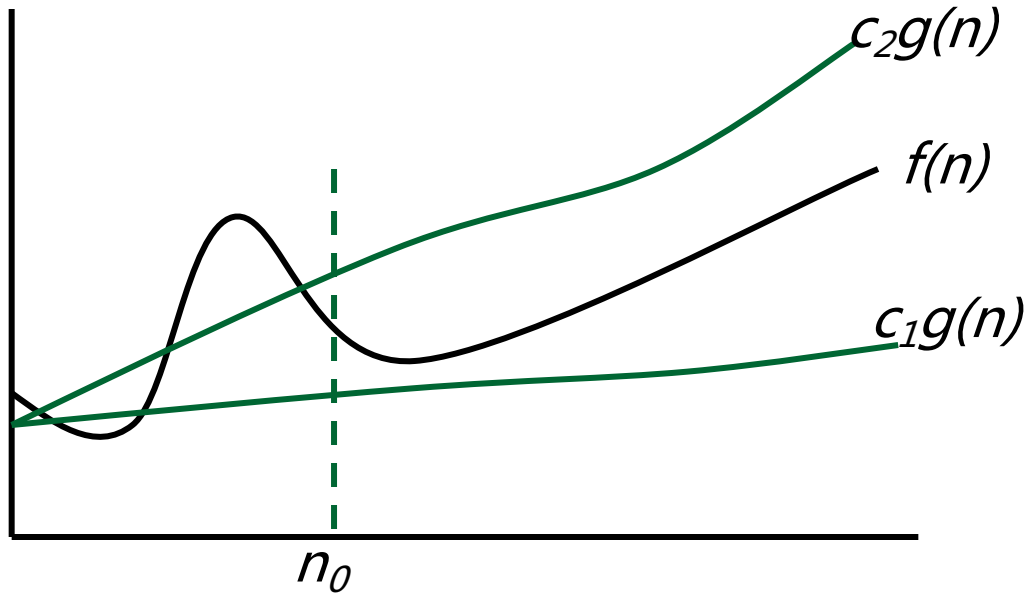
  $$c_1 g(n) \le f(n) \le c_2 g(n)$$

Θ 表式法

$f(n) = \Theta(g(n))$, *iff*

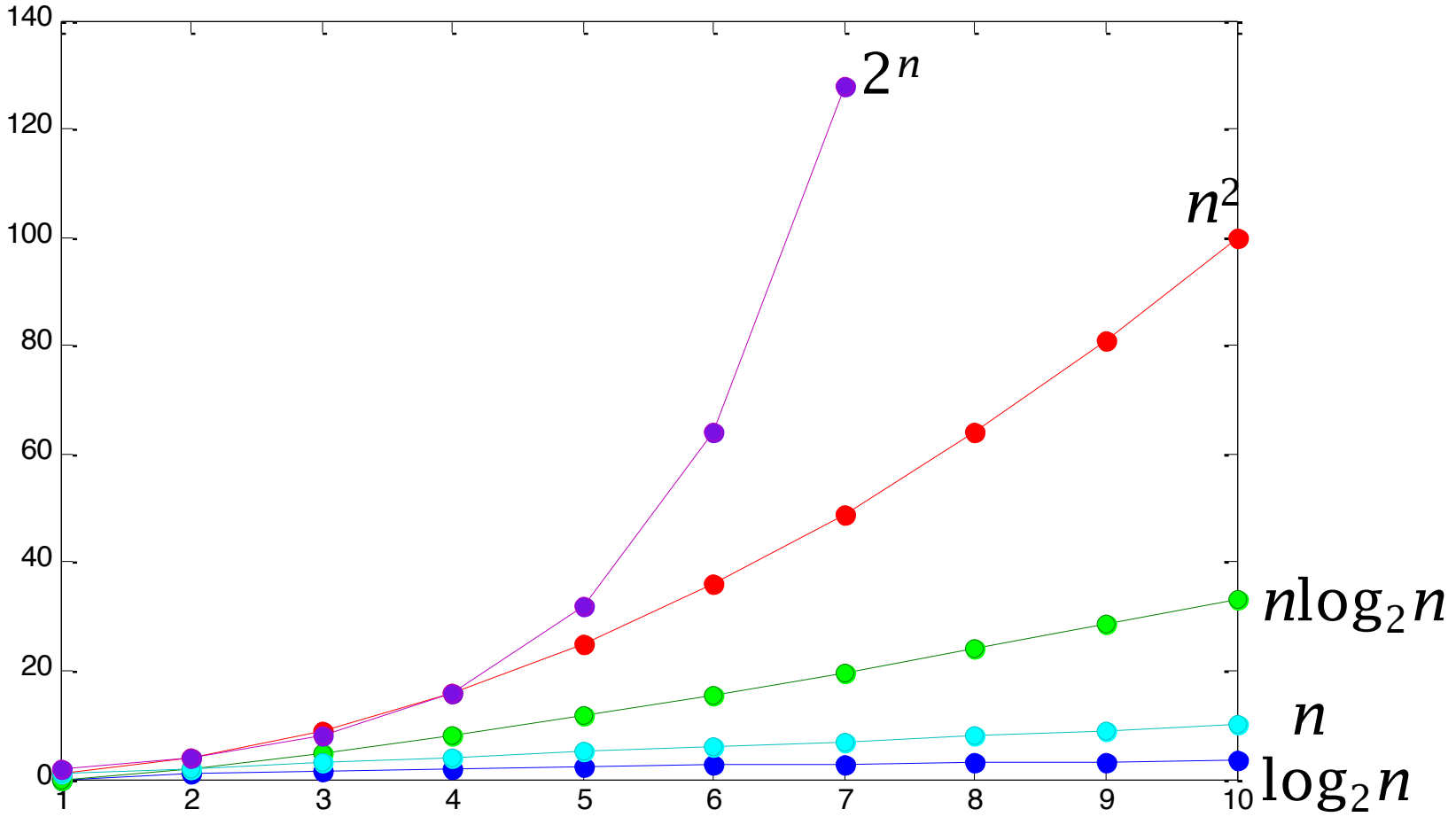
$\exists c_1, c_2, n_0 > 0$ *s.t.*

$0 \le c_1 g(n) \le f(n) \le c_2 g(n), \quad \forall n >= n_0$



f(n) is eventually both lower- & upper- bounded by g(n)

增长率函数曲线



渐近分析示例

对数组中的各个元素求和的代码：

```
for  (i = sum = 0;  i<n;  i++)
        sum += a[i];
```

其中主要的操作为赋值运算，故该算法的时间代价主要体现在赋值操作的数目上

- 在循环开始之前有两次赋值，分别对 i 和对sum进行；
- 循环进行了n 次，每次循环中执行两次赋值，分别对sum和对 i 进行更新操作；
- 总共有2 + 2n 次赋值操作

其渐进复杂度为 $O(n)$

渐近分析示例

依次求出给定数组的所有子数组中各元素之和：

```
for  (i = 0; i < n; i++)
      for  (j = 1, sum = a[0];  j <= i;  j++)
            sum += a[j];
```

- 循环开始前，有一次对 i 的赋值操作；
- 外层循环共进行 n 次，每个循环中包含一个内层循环，以及对 i，j，sum分别进行赋值操作；
- 每个内层循环执行 2 个赋值操作，分别更新sum 和 j；共执行 i 次（i=1,2, …,n-1）
- 整个程序总共执行的赋值操作为：

$$1+3n+\sum_{i=1}^{n-1}2i=1+3n+2(1+2+....n-1)=1+3n+n(n-1)=O(n)+O(n^2)=O(n^2)$$

渐近分析示例

若只对每个子数组的前 5个元素求和，则相应的代码可采用下面的方式：

```
for ( i=4;  i<n;  i++)
    for (j = i-3, sum = a[i-4]; j <= i; j++)
        sum += a[j];
```

- 外层循环进行n-4次
- 对每个 i 而言，内层循环只执行4次，每次的操作次数和 i 的大小无关：8次赋值操作
- 整个代码总共进行O(1)+O(n) + 8(n-4) = O(n)次

看似双重循环，其实线性时间

运行时间估算

- **假设CPU每秒处理10^6 个指令，对于输入规模为10^8的问题，时间代价$T(n) = 2n^2$的算法要运行多长时间?**
  - 操作次数

    $T(n) = T(10^8) = 2\times(10^8)^2 = 2\times10^{16}$
  - 运行时间

    $2\times10^{16}/10^6 = 2\times10^{10}$秒
  - 一天有86,400秒，故需要231,480 天，

    即，634年

运行时间估算

- 假设CPU每秒处理10^6个指令，对于输入规模为10^8的问题，时间代价为 $n\log n$ 的算法要运行多长时间?
  - 操作次数

    $T(n) = T(10^8) = 10^8 \times \log 10^8 = 2.66 \times 10^9$
  - 运行时间为

    $2.66 \times 10^9 / 10^6 = 2.66 \times 10^3$秒，

    即44.33分钟

规定时间内可处理问题规模

- 设CPU每秒处理10^6个指令，则每小时能够解决的最大问题规模

  $$T(n) / 10^6 \leq 3600$$

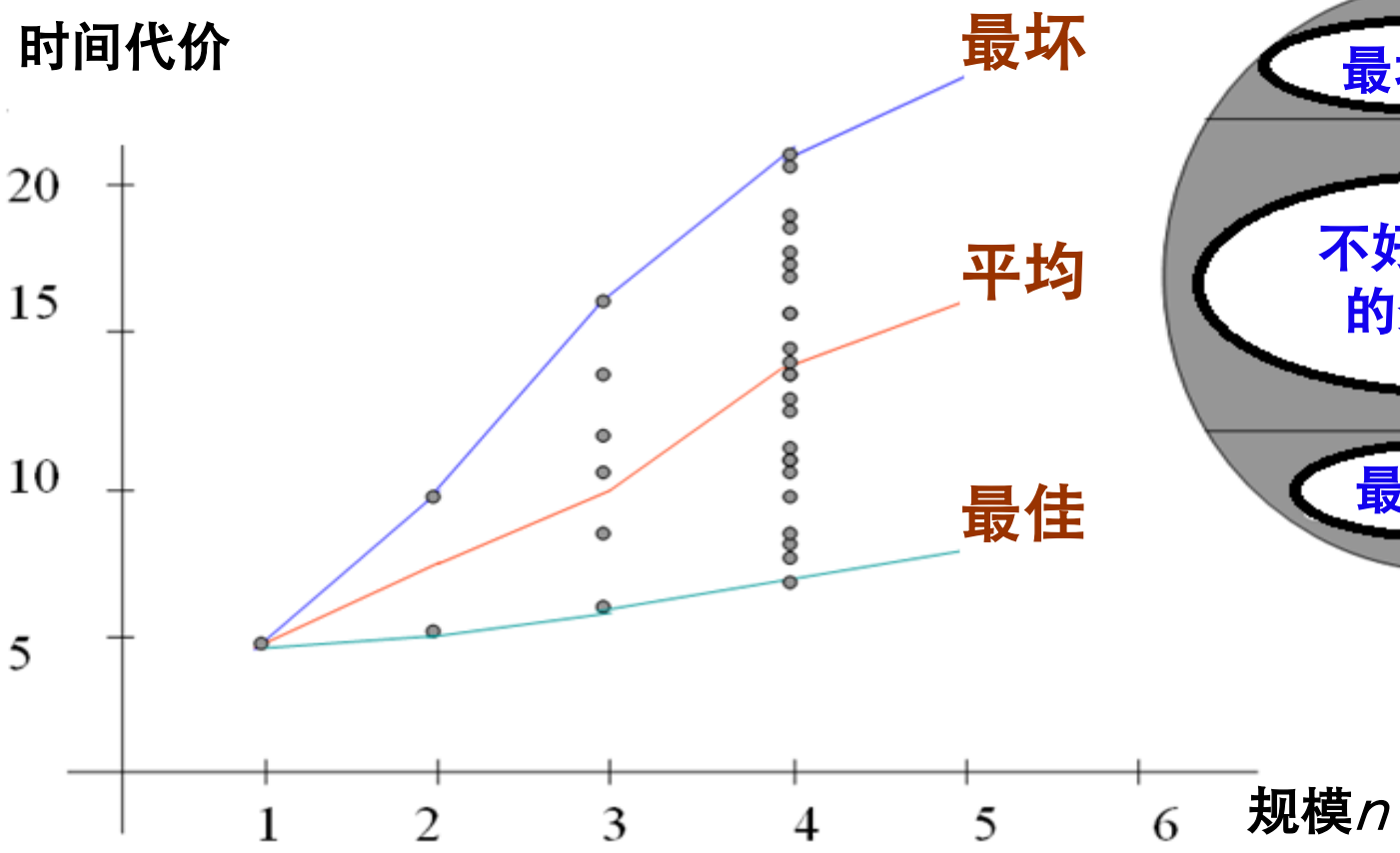
- 对$T(n) = 2n^2$，
  - 即，$2n^2 \leq 3600 \times 10^6$
  - $n \leq 42,426$
- $T(n) = n\log n$
  - 即，$n\log n \leq 3600 \times 10^6$
  - $n \leq 133,000,000$

加快硬件速度?

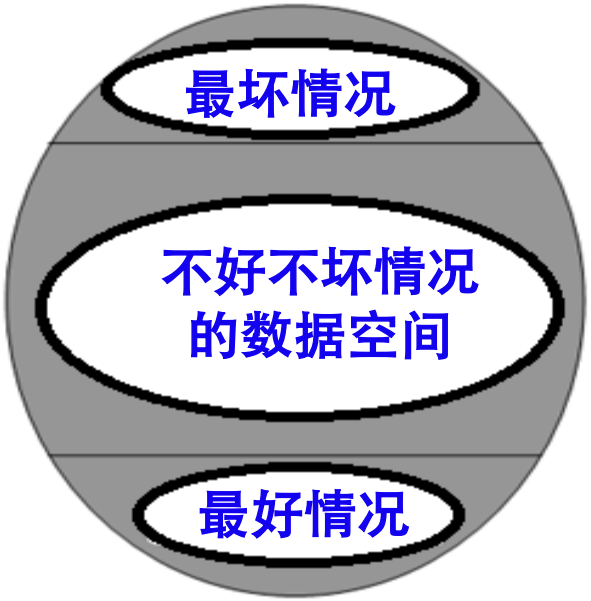
T(n)	处理输入规模为$n=10^8$		1小时内解决的问题规模	
	10^6指令/秒	10^8指令/秒	10^6指令/秒	10^8指令/秒
n log n	44.33 分	0.4433分	1.33亿	100亿
$2n^2$	634年	6.34年	42,426	424,264

- CPU每秒处理10^8个指令（快100倍）
  - 处理时间降为原来的 1/100
  - 解决问题的规模?
    - 对$2n^2$，规模增加10倍
    - 对nlogn，规模增加75倍

最差、最佳、平均



问题的输入数据空间



示例： 顺序找k值

- 顺序从一个规模为n的一维数组中找出一个给定的K值
- **最佳情况**
  - 数组中第1个元素就是K，只要检查一个元素
  - min{ complexity(step(y)) | y∈ Input}
- 最差情况
  - K是数组的最后一个元素，检查数组中所有的n个元素
  - max {complexity(step(y)) | y∈ Input}

示例： 顺序找k值 平均情况

- 等概率分布
  - K值出现在n个位置上概率都是1/n则平均代价

$$\frac{1+2+\ldots+n}{n}=\frac{n+1}{2}$$

- 概率不等
  - 出现在第1个位置的概率为1/2, 第2个位置上的概率为1/4
  - 出现在其他位置的概率都是 $\frac{1-1/2-1/4}{n-2}=\frac{1}{4(n-2)}$
  - 平均代价 $\frac{1}{2}+\frac{2}{4}+\frac{3+\ldots+n}{4(n-2)}=1+\frac{n(n+1)-6}{8(n-2)}=1+\frac{n+3}{8}$
  - Average: $\sum_i p(input_i)steps(input_i)$ $\sum_i P(input_i)=1$

最差、最佳、平均情况

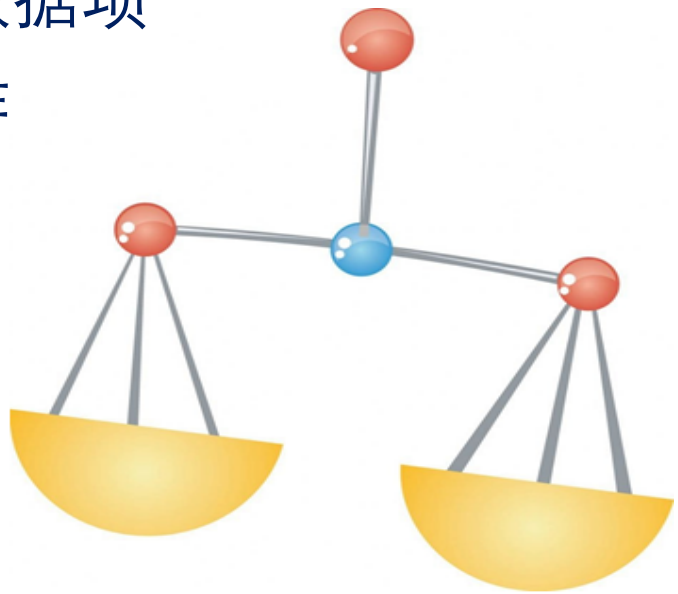
- 就时间开销而言，一般不关心最佳情况，而更关注最差估计，特别是应急事件的处理，计算机系统必须在规定的响应时间内完成
- 对多数算法而言，最差情况和平均情况估计，二者的时间代价公式虽有不同，但往往只是常数因子大小的区别，或者常数项的大小区别，因此不大影响对渐进分析的增长率函数的估计

时间和空间资源开销

- 对于空间开销，也可实行类似的渐近分析
  - 若算法使用的数据结构是静态的存储结构，即存储空间在算法执行过程中并不发生变化
  - 使用动态数据结构的算法，其存储空间在运行过程中是变化的，有时会有数量级的增大或缩小。对于这种情况，空间开销的分析和估计是十分必要的

时间/空间空资源的权衡

- 数据结构
  - 一定的空间来存储它的每一个数据项
  - 一定的时间来执行单个基本操作
- 代价和效益
  - 空间和时间的限制
  - 软件工程



- 时间/空间的折中
  - 增大空间开销可能改善算法的时间开销
  - 节省空间往往需要增大运算时间

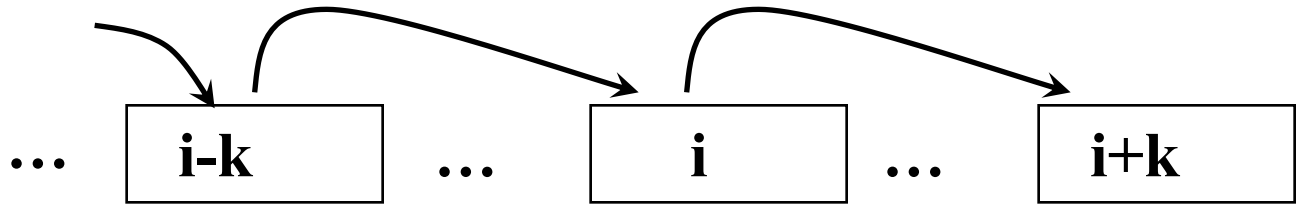
时间/空间空资源的权衡

- 设计一个算法，将数组 A(0..n-1) 中的元素循环右移 k 位，假设原数组序列为 $a_0, a_1, \ldots, a_{n-2}, a_{n-1}$；移动后的序列为 $a_{n-k}, a_{n-k+1}, \ldots, a_0, a_1, \ldots, a_{n-k-1}$。要求只用一个元素大小的附加存储，元素移动或交换次数与 n 线性相关

  例如，n=10, k=3

  原始数组：0 1 2 3 4 5 6 7 8 9

  右移后的：7 8 9 0 1 2 3 4 5 6

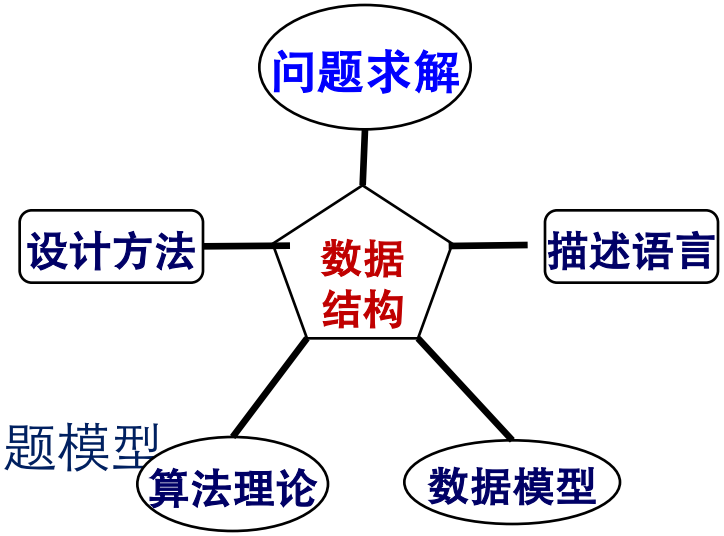


数据结构的选择和评价

- 仔细分析所要解决的问题
  - 特别是求解问题所涉及的数据类型和数据间逻辑关系

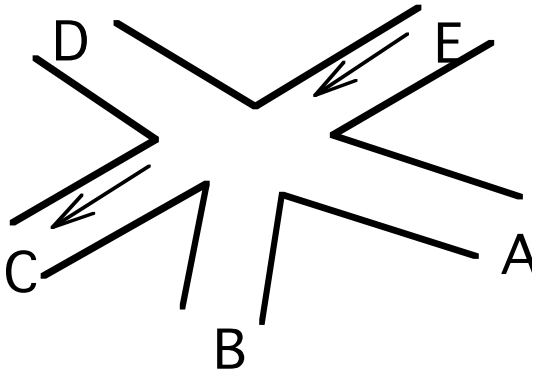
    —— 问题抽象、数据抽象
  - 数据结构的初步设计往往在算法设计之先
- 注意数据结构的可扩展性
  - 考虑当输入数据的规模发生改变时，数据结构是否能够适应求解问题的演变和扩展
- 数据结构的设计和选择需考虑算法的时空开销的优劣及权衡

利用计算机的问题求解



- 问题求解 编程的目标
  - 抽象和建模
    - 问题抽象
      - 分析和抽象任务需求，建立问题模型
    - 数据抽象
      - 确定恰当的数据结构表示数学模型
    - 算法抽象
      - 在数据模型的基础上设计合适的算法
  - 数据结构 + 算法； ➔ 程序设计
    - 模拟和解决实际问题

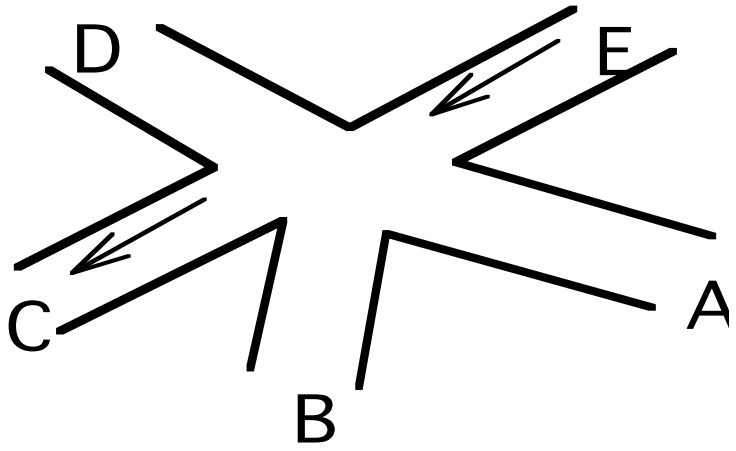
多叉路口交通灯管理问题



- 五叉路口
  - 右行规则
  - 道路C、E是箭头所示的单行道
- 用多少种颜色的交通灯，怎样分配给这些行驶路线?
  - 可同时行驶而不发生碰撞的路线用一种颜色的交通灯指示
  - 颜色越少则管理效率越高
  - 不考虑过渡灯（例如黄灯）

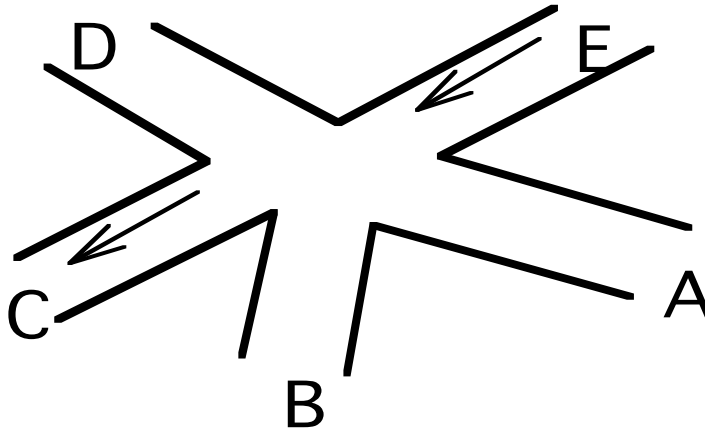
多叉路口交通灯管理问题

- 13种行驶路线
  - AB, AC, AD
  - BA, BC, BD
  - DA, DB, DC
  - EA, EB, EC, ED
- 不能同时, 如
  - AB, BC;
  - EB, AD
- 可以同时，如
  - AB, EC



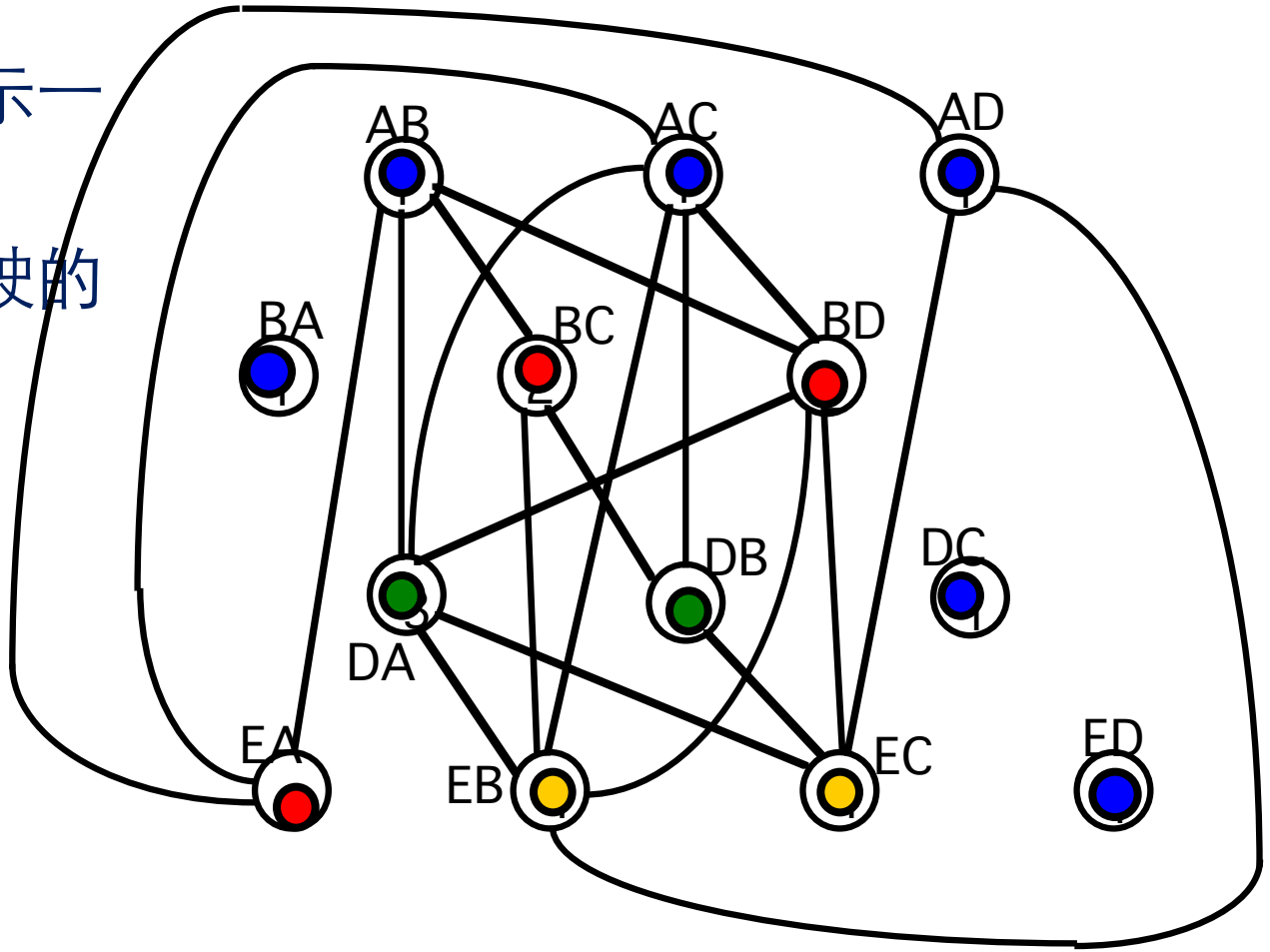
多叉路口交通灯管理问题

- 不能同时走的路线对
  - (AB BC) (AB BD)
  - (AB DA) (AB EA)
  - (AC DA) (AC BD)
  - (AC DB) (AC EA) (AC EB)
  - (AD EA) (AD EB) (AD EC)
  - (BC EB) (BC DB)
  - (BD DA) (BD EB) (BD EC)
  - (DA EB) (DA EC)
  - (DB EC)

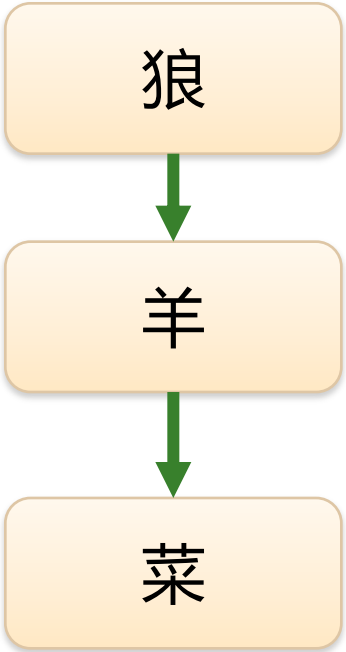


多叉路口交通灯管理问题

- 建模成图
  - 每个顶点表示一种行驶路线
  - 不能同时行驶的点用线连接

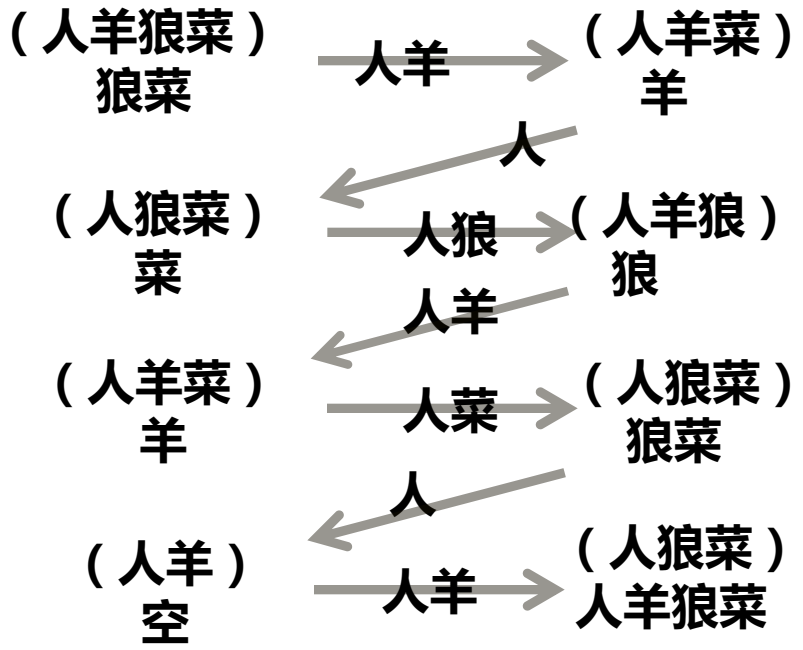


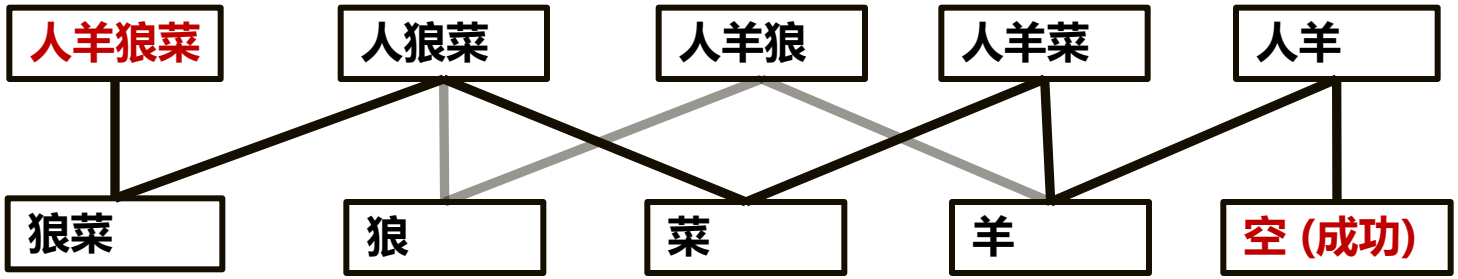
农夫过河



农夫过河

- **问题抽象**：人狼羊菜乘船过河
  - 只有人能撑船
  - 船只有两个位置（包括人）
  - 狼羊、羊菜不能在没有人时共处
- **数据抽象**：图模型
  - 不合理状态：狼羊、人菜、羊菜、人狼、狼羊菜、人
  - 顶点表示“原岸状态”— 10种（包括“空”）
  - 边：一次合理的渡河操作实现的状态转变

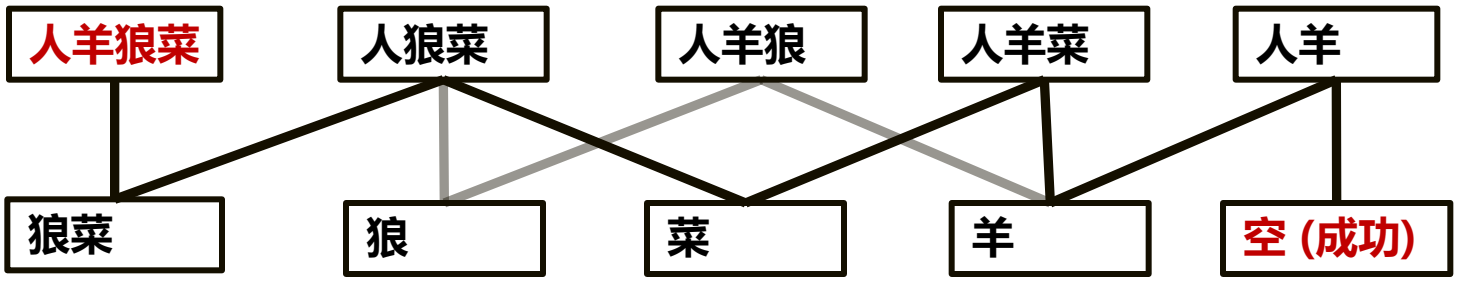




农夫过河

- 数据结构
  - 相邻矩阵
- 算法抽象
  - 最短路径

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$



实践者言

- Rob Pike，C 语言大师之一 ，在 *Notes on C Programming* 中阐述了如下的哲学：
  1. 你无法断定程序会在什么地方耗费运行时间。瓶颈经常出现在想不到的地方，故别急于胡乱找个地方改代码，除非你已证实瓶颈就在那儿
  2. 在你没对代码进行估量，特别是没找到最耗时的那部分之前，别去优化速度
  3. 花哨的算法在 n 很小时通常很慢，而 n 通常很小。花哨算法的常数复杂度很大。除非你确定 n 总是很大，否则避免花哨算法（即使 n 很大，也优先考虑原则 2 ）
  4. 尽量使用简单的算法配合简单的数据结构，花哨的算法比简单算法更易出 bug 、也更难实现。
  5. 数据压倒一切。若已选择了正确的数据结构并把一切都组织得井井有条，正确的算法就不言自明。编程的核心是数据结构，而不是算法
  6. 没有原则 6

小结

- 数据结构的地位与重要意义
- 数据结构的主要内容
- 抽象数据类型的概念
- 算法及其特点
- 算法的有效性度量
- 数据结构的选择

思考题

- **你所认为的信息（information）是什么？应该包括那些成分?**
- **你所认为的计算 （computation）是什么？你最想用计算机来帮助你解决什么样的问题?**