1，引言

- 问题求解
- 数据结构
- 算法
- 算法分析
- Python 程序的代价分析
- 数据抽象：概念和作用
  与过程抽象的比较
- 定义类型（class 定义）

问题求解

- 使用计算机是为了解决实际问题。牵涉到：
  - 能用计算机解决的问题的性质，特点
  - 用计算机解决问题的途径和方法
- 解决一个实际问题，就要在计算机里建立该问题的求解模型：
  - 处理实际问题中的各种对象及其相互关系
    把这方面信息映射到计算机可以处理的表示形式
    用 Python 解决问题，就是映射到 Python 能处理的某种结构
  - 把实际问题的求解过程映射到一个计算过程，用程序实现该过程
    例如，用 Python 语言写出解决问题的程序
- 程序设计/软件开发就是要实现这两个映射
  - 但，应该怎么做？

问题求解

- 为解决一个实际问题而开发程序的工作通常可分成下面四个阶段

  未必能按顺序一次完成，经常需要反复

  回忆上学期课程中画的描述这个过程的图示

1. 分析阶段：弄清需要求解的问题，给出尽可能严格的描述
2. 设计阶段：设计出与实际问题对应的求解方案，进而设计出有关实现的细节方案（信息到数据表示的映射，规划求解过程等）

   这部分工作与本课程关系最密切
3. 编码阶段：用某种计算机可以执行的形式，实现第 2 阶段的设计

   与本课程有关，例如用 Python 编程
4. 测试和维护：确认得到的程序能解决问题，以及为满足某些实际目标或需要而修改程序，扩充功能等

- 下面通过一个例子展示这一过程

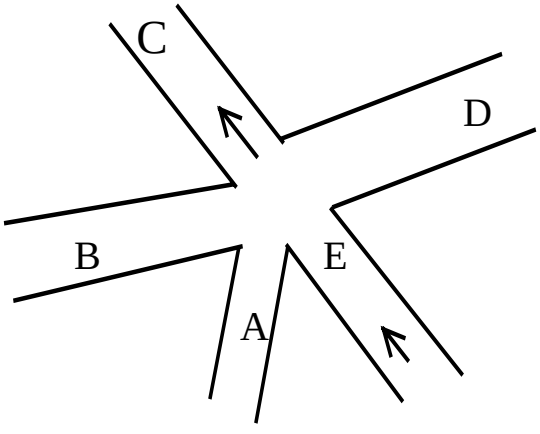
问题求解示例

- 假设现在需要为一个多叉路口设计一个信号灯管理系统
- 具体路口的交通要求情况见图

  图中箭头表示单行方向
- 不同行驶路线间可能出现冲突

  存在现实的安全问题，慎重！
- 需要对行驶方向分组，使：
  - 同属一组的各方向行驶的车辆，同时行驶可以保证安全
  - 分组应该尽可能大一些

    提高路口的效率（经济问题）
- 不是很简单的问题，需要深入分析



一个交叉路口的模型

这个图已经是实际问题的抽象

与正确分配保证安全行车无关的信息已经被抽象掉

集中关注重点是抽象的核心考虑

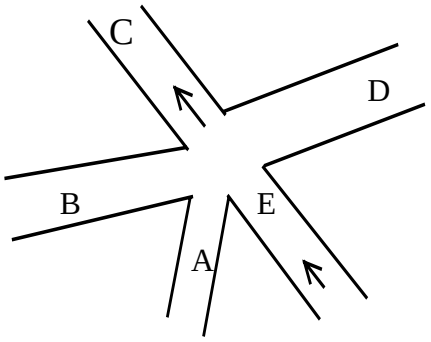
问题分析

- 问题较复杂，要采用某种严格的描述方式，以便能把问题看得更清楚
- 所有可能通行方向（设计一种形式表示）

A→B　A→C　A→D　B→A　B→C

B→D　D→A　D→B　D→C　E→A

E→B　E→C　E→D



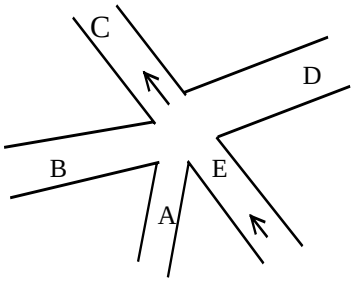
一个交叉路口的模型

- 下面用 **AB** 表示 **A→B**，其他类似
- 考虑如何基于这种抽象表示，进一步看清问题的实质
- 行驶方向的分组，关键情况：
  - 有些方向相互冲突，同时开放会相互阻碍，而且有撞车危险
  - 为了安全，不应该为任意两个冲突的行驶方向同时开绿灯，因此它们不能放入同一个分组

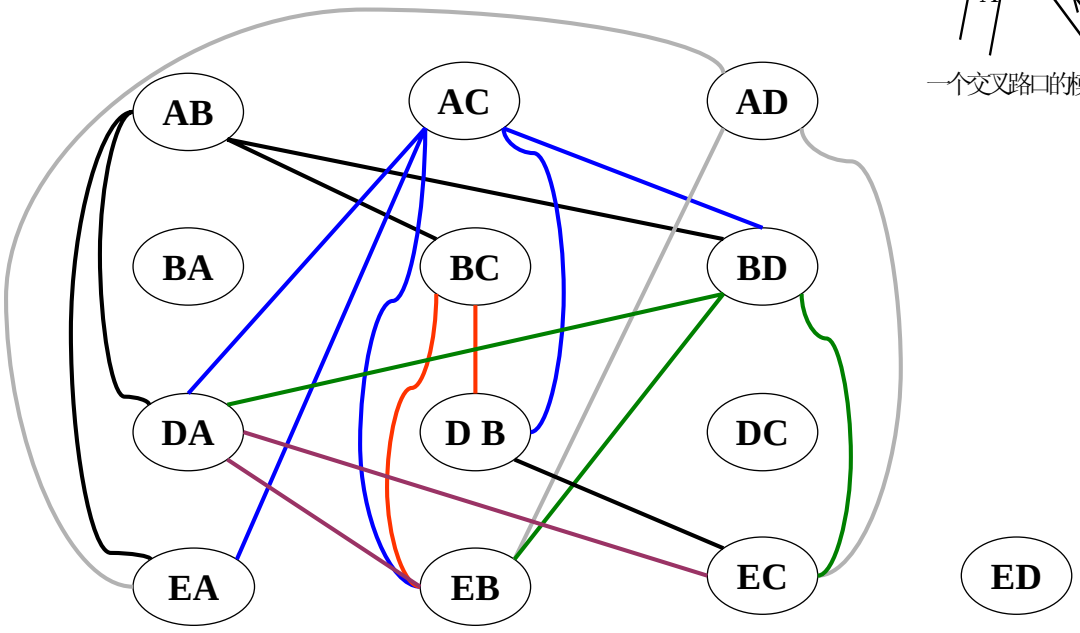
问题分析

- 表示冲突的方式是在冲突方向之间划一条连线

通过认真分析，得到下面的冲突图

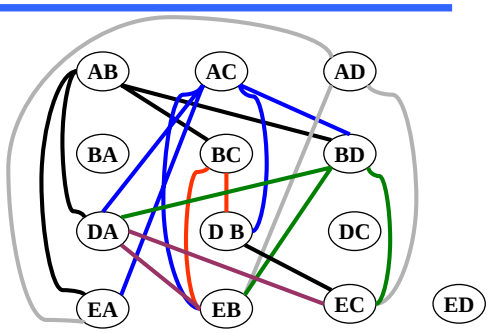


一个交叉路口的模型



问题分析

- 问题求解线索：把冲突图中的结点分组
  - 保证有边相连的结点不同组
  - 同组行驶方向互不冲突，可同时通行
- 问题：哪些结点可同组，共分为多少个组？
  - 显然解：每个方向独立作为一个组
  - 进一步目标：分组最少（同时通行方向多，提高路口利用率）



- 地图着色问题（一个“等价”求解问题）：
  - 把图中结点看作国家，结点间连线看作两国边界
  - 结点问题就变成了著名的“地图着色问题”：求出可将图中所有国家着色，并使相邻国的颜色不同的最少颜色数
  - 由具体问题得到的图（可能）不是平面图（不能画在一个平面上使连线都不相交），因此需要的颜色数可能多于4

算法设计

- 通过分析构造出问题的求解模型后，下步考虑求解算法（的设计）

  算法设计研究求解问题的严格方法

  设计好的算法为编程实现提供了坚实的基础
- 解决本问题（图着色问题）有许多方法

  不同方法有不同的性质，需要考虑
- 方法1，通过穷举选出最优

  设法逐个枚举出所有可能的合法分组，在枚举过程中，记录遇到的最小分组个数和对应的分组情况

  这一过程一定能找到一个“最优”解（分组数最少的解）
- 缺点：可能组合数太多，逐个枚举需要指数时间（算法的效率低）

  如果不同方向的集合比较大，求解时间可能长得无法忍受

  交通路口的支路不多（超过4的情况不多见），可以考虑这一算法

算法设计

- 方法2. “贪心法”

  这是一类典型的算法设计思路（算法的设计模式），基本想法是根据当时掌握的信息，尽可能地向得到解的方向推进

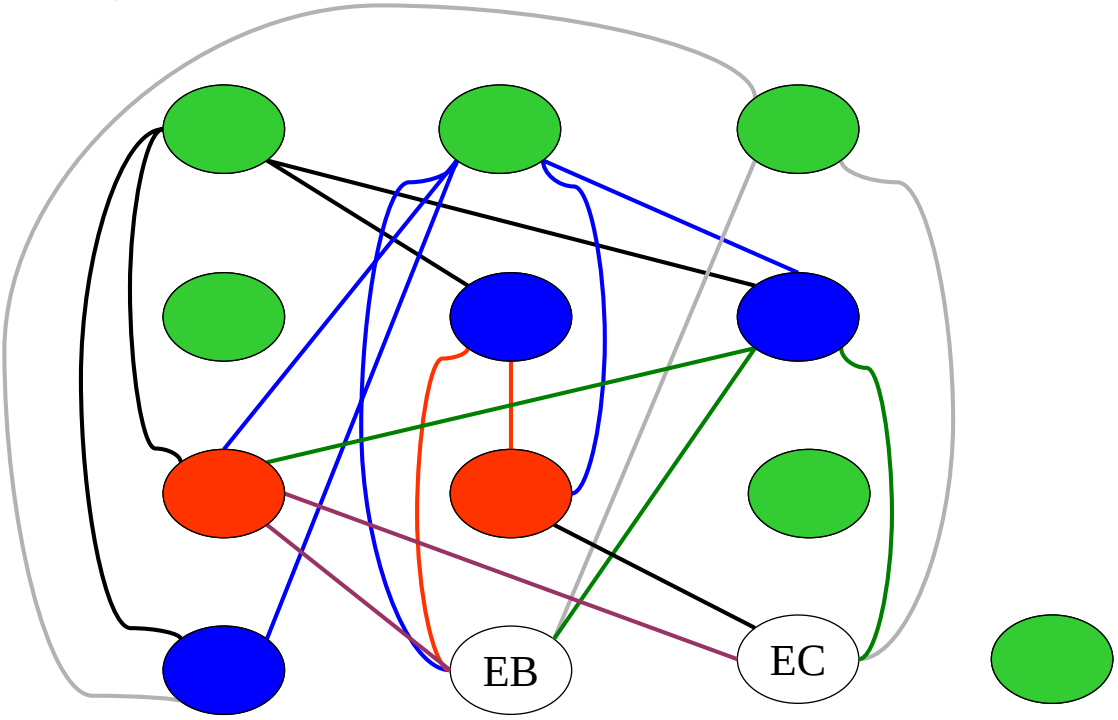
  通常不能找到最优解，但能找到“可接受的”解

- 算法梗概（伪代码）：

```
设法表示图 G                          # 记录图中的结点连接关系
集合 verts 保存 G 中所有结点          # 建立初始状态
设置集合 groups = 空集                # 记录得到的分组，元素是集合
while 存在未着色结点：                # 用贪心法反复找新分组
    选一种新颜色
    在未着色结点中给尽量多无互连边的点着色（构建一个分组）
    记录新着色的结点组
# 算法结束时集合 groups 里记录着一种分组方式
# 算法细节还需要进一步考虑
```


算法设计

- 采用贪心法，按结点排列顺序试探（演示）：



算法设计

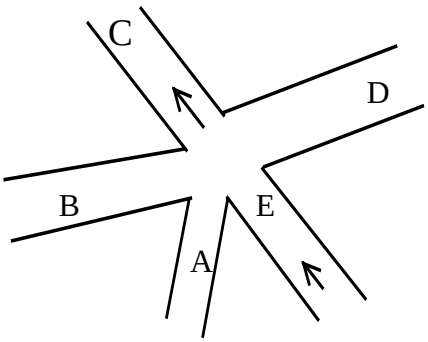
- 贪心法应用于图1.2，得到的分组：

  绿色：AB，AC，AD，BA，DC，ED

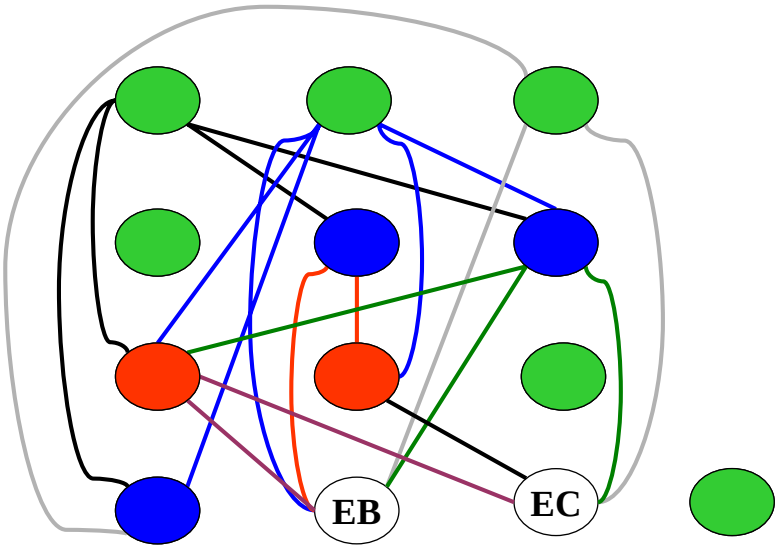
  蓝色：BC，BD，EA

  红色：DA，DB

  白色：EB，EC



一个交叉路口的模型



算法设计

- 考虑算法中缺失的细节：处理一种新颜色的着色
  - 设 G 保存需着色图中结点相互连接的信息，verts 记录图中所有尚未着色结点，初始时 verts 是 G 中所有结点的集合
  - newgroup 记录已确定可用当前新颜色着色的结点（一个集合）
- 找出 verts 里中可用新颜色着色的结点集的程序段框架：

```
newgroup = set()
for v in verts :
    if v 与 newgroup 中所有结点之间都没有边 :
      从 verts 中去掉 v
      把 v 加入 newgroup
    }
# 结束时 newgroup 里是可以用一种新颜色着色的结点
# 用这段代替前面程序框架中主循环体里的一部分
```

算法设计

- 算法的实现要基于一些集合和图操作
- 所需的集合操作（Python 里大都存在）：
  - 判断一个集合是否为空：**vs == set()**
  - 设置一个集合为空：**vs = set()**
  - 从集合中去掉一个元素：**vs.remove(v)**
  - 向集合里增加一个元素：**vs.add(v)**
  - **Python** 的 **set** 不支持元素遍历，现在还要在循环中修改集合。在每次需要遍历时从当时的 **verts** 生成一个表，对表遍历
- 所需要的图操作
  - 检查结点 **v** 与结点集 **newgroup** 中各结点在 **G** 中是否有边连接
  - 操作 **notAdjacentWithSet(v, newgroup, G)** 依赖于图的表示
- 有了上述图、集合和其上的操作，程序实现已经不难了

算法设计

- 还有一些细节：
  - 颜色的表示？可以用顺序的整数
  - 如何记录得到的分组？

    可以把分组作为集合加入 **groups** 作为元素（集合的集合）

    具体颜色实际上不重要，可以记录（如用元组）或不记录
- 这里实际上介绍了两种最基本的算法设计方法：
  - 枚举和选择（选优）
    - 设法枚举所有可能情况，从中找出问题的解（需判断是否为解）
    - 设法枚举所有可能的解，从中找出最优的解（需判断优劣）
  - 贪心法（对复杂问题，可能需要妥协）

    根据已知的局部信息做判断，做出正确的但可能非最优的解

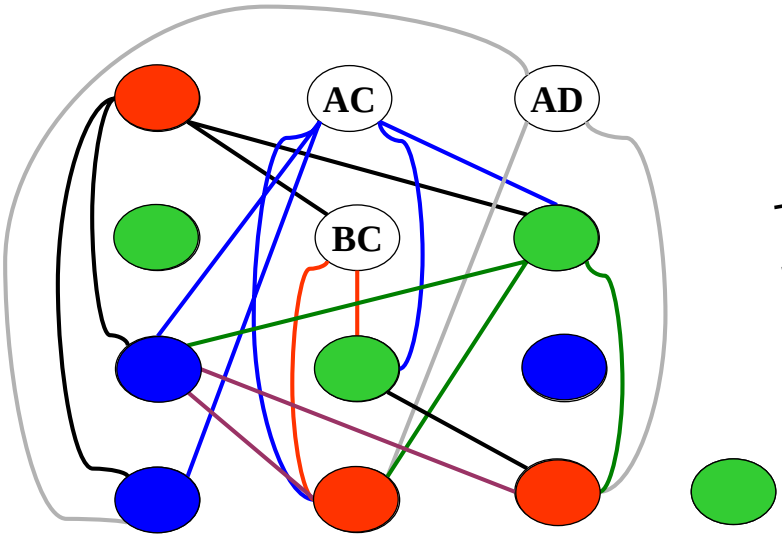
算法设计（精化，refinement）

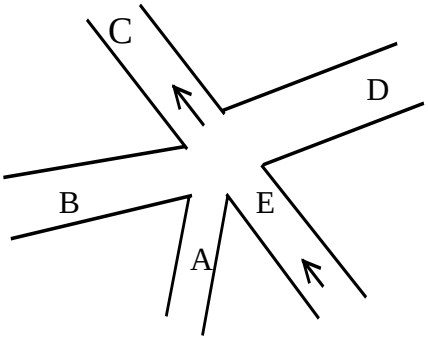
- 贪心着色算法（程序）：

```
def coloring (G) : # 做图 G 的着色
    color = 0
    groups = set()
    verts = vertexes(G) # 取得 G 的所有结点，依赖于图的表示
    while verts != set() :
        newgroup = set()
        for v in list(verts) :
            if notAdjacentWithSet(v, newgroup, G) :
                newgroup.add(v)
                verts.remove(v)
        groups.add((color, newgroup))
        color += 1
    return groups
# 欠缺的细节：图的表示，以及这里涉及的两个图操作
```


分析

- 实际的可能分组不唯一，下面是另一满足要求的分组



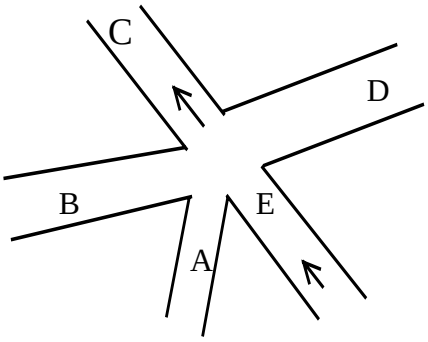


一个交叉路口的模型

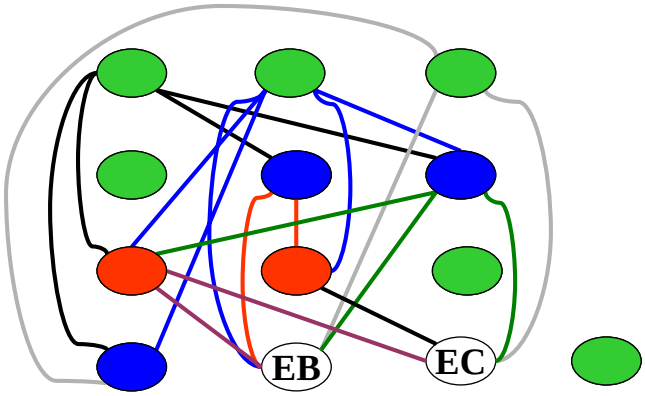
找出最小分组数不是简单工作。已发现的算法都是需要枚举所有组合

实际上，这个问题是 NP 完全性问题

分析



一个交叉路口的模型



解决了原来要解决的问题吗？

得到分组：

1: AB, AC, AD, BA, DC, ED

2: BC, BD, EA

3: DA, DB

4: EB, EC

分析

- 如果希望提供最大行驶可能，问题就不是安全划分，而是基于安全划分的分组最大化。可从划分扩充得到
- 第一种分组的扩充：

  1：AB, AC, AD, BA, DC, ED

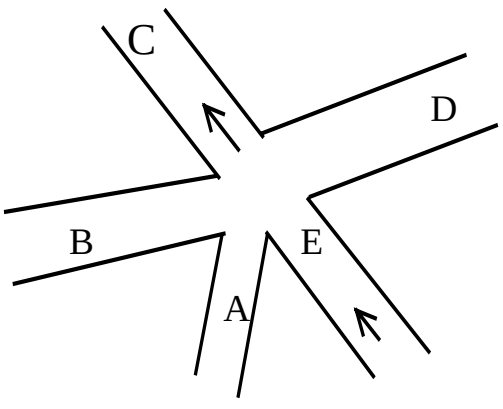
  2：BC, BD, EA, BA, DC, ED

  3：DA, DB, BA, DC, ED, AD

  4：EB, EC, BA, DC, ED, EA

  后两组扩充时可在 **AD** 和 **EA** 任选

- 其他问题：
  - 分组如何更替？
  - 持续时间（公平、实际需要）？
  - … …



一个交叉路口的模型

实例求解小结

- 假设希望做一个程序，给它任一交叉路口信息，就能得到一种可行分组
  - 上面分析已经给出了一种解决问题的方案（算法）
  - 解决问题的下一步是做出程序。怎么做？
- Python 为这一算法的实现提供了结构和操作，但也缺一些
  - 有集合和集合操作，但没有图及所需操作
  - 设法实现图的所需功能，前面算法就可以进一步转化为实际程序
- 有些语言没有这些高级结构，只有一组基本类型和几个数据组合机制
  - 例如 C，只有几个基本类型，和数组/结构/指针等低级组合机制
  - 这时就需要自己实现集合、图及其相应操作
- 理解和有效实现这类高级结构是“数据结构”课研究的问题

  还能帮助理解 Python 各种结构的性质和合理使用方法

问题和算法

- 本课程只考虑通过计算解决的问题，将其简称“问题”
- 对一个给定问题，用某种严格方式描述一个求解过程，对该问题的每个实例，该过程都能给出解，这个描述就是解决该问题的一个算法
- 上学期的例：求任意实数 x 的平方根，是一个问题
  - 下面描述不是算法：

    取得那个大于等于 0 且使 $y^2 = x$ 的 y

    它没给出一个可以按部就班遵照执行的求解过程
  - 下面描述是一个算法（牛顿迭代法）：

    1. 任取一个非 0 的初始值 y（例如取 y = 1）
    2. 如果 y^2 与 x 足够接近，结束并把 y 当作 x 的平方根
    3. 取 z = (y + x/y) / 2
    4. 令 y 取 z 作为新的值，回到步骤 2

问题和实例

- 一个“问题”是具有同样性质的实例的集合。例如
  - 判断一个数是否素数，判 **123** 或 **88793767** 是否素数是其实例
  - 求两个（两维）矩阵的乘积
  - 将一个整系数多项式分解为不可约整系数多项式因子的乘积
  - 将一个图像旋转 **90** 度
  - 辨识数字相机取景图像里的人脸
- 可能用一个算法统一地求解一个问题的（所有）实例，为此需要
  - 设计问题实例的表示
  - 设计一个求解算法
  - 使用算法就是把问题实例（的表示）送给它，得到相应的解
- 有的问题（可以严格描述）没有算法（不可计算问题）；有的问题有算法。对一个问题，可能有不同的算法

算法（Algorithm）

- 算法是问题求解过程的精确描述，具有如下性质：
  - 有穷性（描述的有穷性）：由有限条“指令”/“语句”构成
  - 能行性：指令（语句）含义简单明确，其过程可以完全机械地进行
  - 确定性：作用于所求解问题的给定输入（要处理的问题实例的某种描述），将产生出唯一的确定的动作序列

    确定性算法

    也可以考虑更广泛的概念，如非确定性算法
  - 终止性（行为的有穷性）：产生的动作序列有穷，它或终止并给出问题的解；或终止并指出对给定的输入本问题无解

    也存在不要求终止的计算描述，或称为“过程 **(pocedure)**”
  - 输入/输出：有确定的输入和输出

算法的描述

- 算法可以用不同的方式描述

  需要在易读易理解和严格性之间取得某种平衡

- 用自然语言描述的计算过程（可能易读，但可能出现歧义）

  如在自然语言描述中结合一些数学形式的描述（减少歧义）

- 采用严格的形式化记法形式描述。例如
  - 用图灵机模型描述，定义完成该计算的图灵机（可能极不易读）
  - 用某种严格的形式化算法描述语言描述算法
- 采用伪代码形式，结合严格描述和自然语言
  - 用类似程序语言的方式描述算法过程，其中用一些数学符号和记法描述细节和操作（如前面平方根算法描述用了控制转移结构）
  - 例如，用 **Python** 语言结构，结合自然语言的局部功能说明

算法和程序

- 一个算法，描述了一个问题的解决过程，通常主要供人看，供人思考和理解相应的问题求解方法、技术和过程
- 人可以按算法一步步工作，完成具体问题实例的求解

  例如，按前面算法，求出 **2** 的精确到 **5** 位小数的平方根

- 计算问题通常很复杂，人工做不现实，只能处理很简单问题的规模很小的实例。利用自动化的计算机器完成复杂计算工作，才可能解决有实际价值的问题。今天，就是要指挥计算机完成计算
- 程序：用计算装置能处理的语言描述的算法。例如，**Python** 语言描述的程序。程序是算法的实现
- 程序可能用各种计算机语言描述，例如
  - 用直接对应特定计算机硬件的机器语言或汇编语言
  - 用通用的编程语言，如 **C**、**Java** 等
  - 我们将用 **Python** 语言

算法和程序

- 算法和程序密切相关
  - 每一个程序的背后，都隐藏着一个或一些算法

    正确实现的程序能解决相关算法所解决的问题，其（运行时的）动态性质反应了相关算法的特征（是相应算法的合理实现）
  - 程序用某种计算机能处理的具体编程语言描述，通常会包含一些与具体语言有关的细节结构和描述方式方面的特征
- 在抽象考虑一个计算过程，或考虑该计算过程的抽象性质时，人们常用“算法”作为术语，指称相应计算过程的描述
- 在考虑一个计算（在某种语言里）的具体实现和实现中的问题时，人们常用“程序”这一术语讨论相关问题
- 下面讨论中也采用这种说法
  - 有时我们写一个程序，但讨论时却说“算法”。实际上是指该程序背后的与具体语言无关的计算过程

算法的设计

- 实际应用中的算法，表现形式千变万化
  - 但是许多算法的设计思想有相似之处
  - 可以对它们分类，进行学习和研究（后面例子和讨论）
- 设计算法的一些核心的通用想法可以称为算法设计模式。常见：
  - 贪心法
  - 分治法
  - 回溯法（搜索法）
  - 动态规划法
  - 分支限界法
- 对于算法，没有放之四海而皆灵的设计理论或技术，只能借鉴

  算法是智力活动的产物，一个好算法至少等价于一个好定理和证明

算法分析

- 算法需要实际使用（转变为程序），为此需要理解
  - 需要考察算法的性质，比较算法的优劣，理解算法的适应范围
  - 算法分析：度量算法性质的工作
- 最常需要考虑的特性是算法的时间和空间开销（因为要实际使用）
  - 度量一个算法的开销，需要有合理统一的标准
  - 处理问题的小实例和大实例，开销通常不同
    - 开销通常都依赖于问题实例的规模
    - 算法所用时间和空间通常会随着实例规模的增大而增长
  - 即使规模相同，处理不同实例的代价也可能不同。对两个算法，
    - 可以考虑其解决同样规模实例的平均代价
    - 也可以考虑其解决同样规模实例的最大代价（最坏情况）

计算代价的意义

- 在考虑求解一个问题的具体算法时，需要考虑用它解决问题的代价
  - 求解过程中需要多少存储空间（需关心存储占用的高位限）
  - 完成问题实例的求解需要多少时间
- 对算法性质的认识非常重要
  - 比较解决同一问题的不同算法。一般说，使用资源少的算法更可取
  - 估计一件计算工作完成的可能性和时间：具体机器的存储量是否够用；在实际所要求的时间内能否完成
- 例如：
  - 做天气预报的程序，必须今天下午完成明天的预报计算
  - 数字相机的人脸识别程序，必须在几分之一秒完成工作。用户不会接受更慢的算法。对这类问题，如果找不到效率够高的好算法，可以考虑不那么准确但更快速的算法

问题实例规模与算法代价

- 问题实例通常可以基于某种规模度量，反映实例大小，决定计算开销
  - 被判断素数性的整数的大小（或者数的长度）
  - 求乘积的两个矩阵的各个维的长度
  - 多项式的次数（或者长度）
  - 被旋转或被辨识的图像的大小（长和宽的像素数，或它们的乘积）
- 显然，一般而言，求解规模大的实例要付出更高的代价
  - 考虑具体求解算法的代价时，应该用问题实例的规模作为参数
  - 也就是说，算法的代价是用它求解的问题实例规模的函数
- 例：求斐波那契数列 F_i 的一个算法，定义是

  $F_0 = F_1 = 1$

  $F_n = F_{n-1} + F_{n-2}$　　对于 $n > 1$

求斐波那契数（1）

直接根据数学定义写出的算法（用 Python 描述）：

```
def fib(n) :
    if n < 2 :
        return 1
    else :
        return fib(n-1) + fib(n-2)
```

把参数 n 看作实例规模，可知计算 F_n 的时间代价（操作次数）大致等于计算 F_{n-1} 和 F_{n-2} 的时间代价之和

这一代价大约等于斐波那契数 F_n 的值，而且已有结论

$$\lim_{n\to\infty} F_n = \left(\frac{\sqrt{5}+1}{2}\right)^n$$

括号里表达式 ≈ 1.618，所以计算 F_n 的时间代价按 n 的值指数增长。对较大的 n，这一计算需要很长很长时间

求斐波那契数（2）

- 求斐波那契数的另一个算法：

  对 F_0 和 F_1 直接给出结果

  否则从 F_{k-1} 和 F_{k-2} 计算 F_k，直至 k 等于 n 时给出 F_n

- 对应 Python 实现：

```
def fib(n) :
    f1 = f2 = 1
    for k in range(1, n) :
        f1, f2 = f2, f2 + f1
    return f2
```

- 计算 F_n 的值，循环前工作做一次，循环做 n – 1 次。每次循环执行几个简单动作，总工作量（基本操作执行次数）与 n 值成某种线性关系
- 这两个例说明，解决同一问题的不同算法，计算代价的差异可以很大，甚至性质截然不同。这也说明了分析算法（程序）复杂性的意义

算法的时间和空间代价

- 空间代价

  被解决实例的规模（以某种单位计量）为 n 时，求解算法所需要的存储空间按某种单位为 S(n)，称该算法的空间代价为 S(n)

- 时间代价

  被求解实例的规模为 n 时，求解算法所耗费的时间以某种单位计算为 T(n)，称该算法的时间代价为 T(n)

- 与度量有关的三个概念（需要根据实际问题确定）：

  问题规模，空间单位，时间单位

- 对一个具体程序（算法的实现），可能给出代价的精确估计
  - 可能准确统计执行中所做的各种基本操作（作为规模 n 的函数）
  - 如果程序在具体计算机上运行，而且已知各基本操作所需时间，就可能算出程序执行所需的时间（也是规模 n 的函数）

算法的时间和空间代价

- 对抽象算法，通常无法做精确度量

  只能退而求其次，设法估计算法复杂性的量级

- 例：求两个 n×n 矩阵乘积的常规乘法算法

  主要运算：乘法（或加法）

  空间单位：一个元素所占存储 s；

  时间单位：一次乘法所用时间 t

  实例规模：矩阵一行（一列）的元素个数

  时间复杂性：$f(n) \approx n \times n \times n = n^3$（单位为 t）

  空间复杂性：只考虑完成算法所需空间，包括结果的存储空间和计算中临时使用的辅助空间，这里约为 n^2（结果矩阵所需空间）

- 下面主要讨论时间代价，空间代价的讨论与此类似

算法的时间和空间复杂性

- 考虑时空代价时，有些因素的准确值意义不大，例如：
  - 时间或空间的基本单位
  - 算法描述和实现的细节差异
- 人们最关注的是算法代价的关键情况和趋势，排除各方面具体细节
  - 从这种看法出发，人们定义了算法 “复杂性”的概念
  - 同样，可以考虑时间和空间复杂性
- 在理论上考虑复杂性时，通常忽略常量因子
  - 例如，代价为 $3n^2$ 和 $100n^2$ 的算法，看作复杂性相同的算法
  - 如果算法改进只是减小常量因子，从理论上看其复杂性没变，但在实际中可能有意义

    例：3 天算出明天的天气预报，与半天算出明天的天气预报

平均和最坏情况

- 对同一问题的同样规模的实例，算法计算的代价也未必一样
- 例如：在任意一个整数序列（如 **Python** 的 **list**）里找出第一个小于 **0** 的整数的位置，找不到时给出一个特殊值（例如 **–1**）

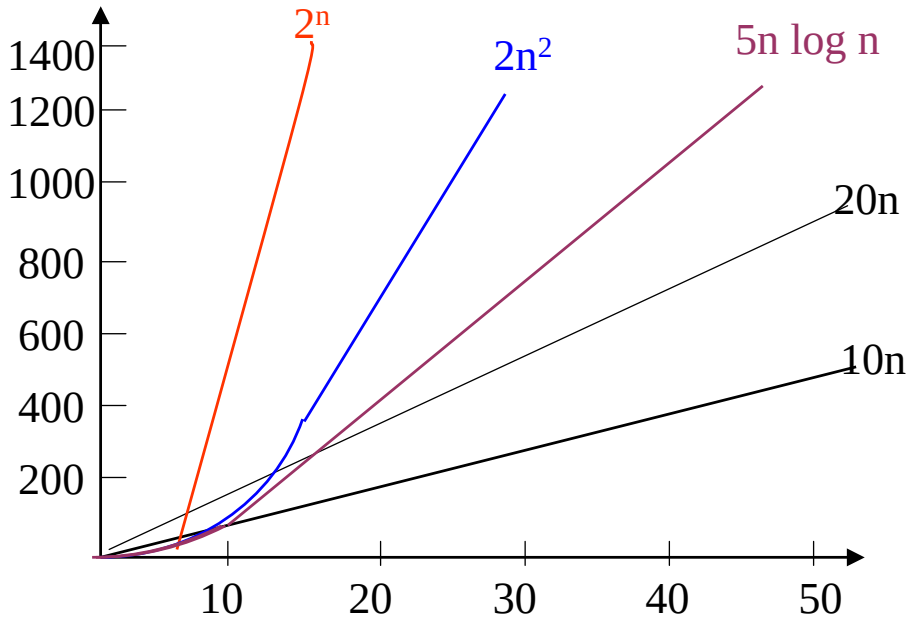
  算法：顺序检查序列里的整数。将表大小 **n** 作为实例规模（很合理），对规模为 **n** 的实例，可能出现多种不同情况

  1. 如果表中第一个元素就小于 **0**，计算中只需比较 **1** 次
  2. 如果表中没有这样的整数，需要比较 **n** 次后才能给出结论
  3. 其他情况下的比较次数在 **1** 和 **n** 之间
- 情况 **1** 是最好情况，意义不大（世界上没有那么多好事）。情况 **2** 是最坏情况，给出了保证：算法在该时间期限内一定能完成指定工作。这称为“最坏情况时间复杂性”。还可以考虑“平均情况时间复杂性”，这种复杂性依赖于实例的分布，需要做假设，而且不易计算
- 我们主要关心最坏情况，有时关心平均情况

大 O 记法

- 算法的复杂性是算法实际代价的抽象
  - 引进一些记法，简化对算法复杂性的描述
  - 记号不太统一，但大 **O** 记法一般都采用
- 定义：如果存在两个正常数 **c** 和 **n0**，当实例的规模 **n** ≥ **n0** 后，某算法的时间（或者空间）代价 **T(n)** ≤ **c** · **f(n)**（或 **S(n)** ≤ **c** · **f(n)** ），则说该算法的时间代价（或者空间代价）为 **O(f(n))**
  - 这个定义就是说，当实例的规模 **n** 充分大时，该算法所需时间（空间）不大于 **f(n)** 的某个常数倍
  - 也说该算法的时间（或空间）代价的增长率为 **f(n)**
- 大 **O** 记法表示算法复杂性的上界（的量级）
  - 还可考虑下界、上确界、下确界。有时不容易得到确切的复杂性
  - 这类记法也用在对“问题的复杂性”的讨论中。后面简单介绍

复杂性增长的阶



O(1) < O(logn) < O(n) < O(n log n) < $O(n^2)$ < $O(2^n)$

常数　　对数　　线性　　**n log n**　　平方　　指数

算法复杂性

- 大 **O** 记法表示一个算法的复杂性的上界
  - 显然，一般性的上界没什么意义（可以任意大）
  - 最希望得到算法复杂性的上确界（的量级），即算法（遇到最难做的实例时）的最坏情况能达到的上界。但常常不容易确定
  - 如不能得到上确界，也希望得到算法的尽可能紧的上界
  - 下面讨论中总用大 **O** 记法表示这样的上界
- 常量时间的复杂性用 **O(1)** 表示，线性复杂性是 **O(n)**，等等
- 算法分析研究推导算法复杂性的技术，其主要技术是构造和求解递归方程。后面将简单介绍有关情况
- 本课程讨论的算法，从结构上看都比较简单
  - 一般不需要高级的分析和推导技术
  - 分析时间复杂性只需要几条最基本的计算规则，后面介绍

算法的复杂性

- 易见，算法的复杂性分级就是（数学分析里）无穷大的阶：

  在规模 n 趋于无穷大的过程中，算法的开销增长的速度

- 算法复杂性高，其代价随规模增大而增长的速度快。重要吗？
- 例：设解决某具体问题的基本操作每秒做 10000次，实例规模是 100，
  - O(n) 的算法，所需时间可忽略不计（1/100秒）
  - $O(n^3)$ 的算法，所需时间是分钟的量级
  - $O(2^n)$ 的算法，所需时间是 $4.0*10^{18}$ 年的量级。（迄今为之的宇宙寿命估计为 10^{10} 年的量级）
- 算法的复杂性反过来决定了算法的可用性：
  - 如果一算法的复杂性较低，就可能用它去解决很大的实例
  - 如果一算法的复杂性高，它只能用于很小的实例。可用性低

复杂性的计算（推导/估计）

现在考虑算法复杂性的“计算”（推导）

循环算法的时间复杂性计算规则

1. 加法规则（顺序复合）

如果所考虑的算法（或其中片段）分为两个部分（或者多个部分），其复杂性是这两部分（或多部分）的复杂性之和

$$T(n) = T1(n) + T2(n) = O(T1(n)) + O(T2(n))$$

$$= O(max(T1(n), T2(n)))$$

由于忽略常量因子，加法等价于求最大值

2. 乘法规则（循环）

如果算法中循环执行 T1(n) 次，每次循环用 T2(n) 时间，则

$$T(n) = T1(n) \times T2(n)$$

$$= O(T1(n)) \times O(T2(n)) = O(T1(n) \times T2(n))$$

复杂性计算实例

- 例：矩阵乘法，求两个 n×n 矩阵 m1, m2 的乘积 m

  假设矩阵实现为两层的表，已准备好保存结果的 m

```
for i in range(n) :
    for j in range(n) :
        x = 0
        for k in range(n) :
            x = x + m1[i][k] * m2[k][j]
        m[i][j] = x
```

- 复杂性的计算，以矩阵的维数作为 n：

  $T(n) = O(n)\times O(n)\times(O(1)+O(n)))$

  $= O(n)\times O(n)\times O(n) = O(n\times n\times n) = O(n^3)$

  这个算法需要用一个 n×n 的结果矩阵，只使用了一个辅助变量（空间是常量），算法的空间复杂性是 $O(n^2)$

复杂性计算实例

- 问题：求 n 阶方阵的行列式的值
- 高斯消元法（一种算法）：
  - 一列消元需要做 $O(n^2)$ 次乘法和减法
  - 整个算法的复杂性是 $O(n^3)$
- 采用行列式求值的定义

  求 n 阶行列式的值需要构造和计算 n 个 n-1 阶行列式

$$\begin{aligned} T(n) &= n \times ((n-1)^2 + T(n-1)) \\ &> n \times T(n-1) \\ &> n \times (n-1) \times T(n-2) \\ &= O(n!) \end{aligned}$$

  $O(n!)$ 是比 $O(2^n)$ 更高的复杂性。

  显然，这种算法没有太大实际意义，对很小的 n 就等不到计算结果了

递归算法的复杂性

- 对比较规范的递归算法，有清晰的理论分析方法

  前面行列式求值的递归算法的情况更复杂一些

- 设一个递归求解算法，将规模为 **n** 的问题实例归结到 **a** 个规模为 **n/b** 的子问题，每次递归时还需要做 $O(n^k)$ 的其他工作，那么

$$T(n) = a \cdot T(n/b) + O(n^k)$$

求解这一递归方程，可以得到下面结果：

当 $a > b^k$ 时， $T(n) = O(n^{\log_b a})$

当 $a = b^k$ 时， $T(n) = O(n^k \cdot \log n)$

当 $a < b^k$ 时， $T(n) = O(n^k)$

注意：这里的 **a**、**b** 和 **k** 是常量，可以覆盖大部分典型情况

对前面行列式求值的示例，上面的解无效（上面公式中 **a** 和 **b** 为常量）

Python 程序的计算复杂性问题

- **Python** 程序是算法的实现，因此也可以考虑其复杂性问题
- 需特别注意：**Python** 的很多基本操作不是常量时间的
  - 基本算术运算是常量时间操作
  - 组合对象操作有些是常量时间的，有些不是。例如：
    - 复制和切片操作，通常需要线性时间（**O(n)** 操作）
    - 表和元组的元素访问和元素赋值，是常量时间的
- 处理组合对象时需要特别考虑操作的复杂性问题
  - 创建对象，需要空间/时间（都是线性复杂性，与对象大小相关）
  - 内置复合数据类型的一些操作的情况后面讨论
  - 用 **Python** 语言，很容易写出一些貌似正确但实际上完全不能用的程序（复杂性太高，而且毫无必要）
  - 今后写程序要强调这点。作业里每个程序都要做复杂性分析

Python 程序的计算复杂性

- 在下面有关数据结构和算法的讨论中，将会介绍和分析一些 **Python** 结构和操作的效率问题
- 一些重要问题（上学期有同学问到其中的一些问题）：
  - 构造新结构，如 **list, set** 等

    显然，构造一个包含 **n** 个元素的结构，至少需要 **O(n)** 时间
  - **list** 操作的效率，如

    元素访问和修改，加入删除元素等
  - 字典 **dict** 操作的效率

    加入新的关键码-值对，关键码查找等
- 程序里经常用到这些操作
  - 它们的效率对程序效率有重大影响
  - 有些操作效率更高，应该优先选用

实例

- 考虑一个简单例子：假设要把得到的一系列数据存入一个表，假设得到一个数据是 **O(1)** 常量时间操作。可以写

```
data = []
while 还有数据：
    x = 下一数据
    data.insert(0, x)  # 把新数据加在表的最前面
```

或者写

```
data = []
while 还有数据：
    x = 下一数据
    data.insert(len(data), x) # 把新数据加在表的最后面
```

- 上面两种写法效率怎样？应该与表的结构和操作的实现方式有关
- 实际情况：前一写法 **O(n^2)** 复杂性，后一个是 **O(n)** 复杂性，与表长度有关。与 **list** 的实现方式有关

实例

- 考虑下面几个函数：

```
def test1(n):
    lst = []
    for i in range(n * 10000):
        lst = lst + [i]

def test2(n):
    lst = []
    for i in range(n * 10000):
        lst.append(i)

def test3(n):
    lst = [i for i in range(n * 10000)]

def test4(n):
    lst = list(range(n * 10000))
```

完成同一工作的不同方式。请自己试验看其代价随 n 增长的趋势。

效率陷阱

- 用 Python 等高级语言编程，存在一些“效率陷阱”
  - “效率陷阱”使原本有可能用计算机做的事情变得“不行了”（时间代价太大）。至少也浪费了计算机和人的大量时间
  - 这种缺陷很可能葬送一个软件，至少损害其可用性（降低其价值）
- 另例见于一些同学上学期的练习（包括期末考试），常见的
  - 程序里以递归的方式构造出一些复杂的结构（如 list 等），而后只使用其中的个别元素
  - 局部看，是使常量时间操作变成了线性时间操作；从递归算法的全局看，经常使多项式时间算法变成指数时间算法
- 这些例子说明了一个情况：在 Python 这样的系统里有许多高级数据机制可用，但要有效使用，有必要了解数据结构的一些深入情况
- 此外，一些同学也可能有兴趣知道 Python 这样强大的系统是如何构造起来的。本课程也能给大家一个理解

问题和复杂性

- 对一个具体的问题，可能存在许多解决它的算法
  - 不同算法的复杂性可能不同
  - 解决问题的算法的复杂性反过来刻画了问题的性质
- 理论观点：解决一个问题的复杂性最低的算法，刻画了这个问题的本质（问题的难度，问题的复杂性）
  - 如果一个问题有复杂性较低的算法，它就是“较易”求解的问题
  - 有关问题的复杂性的研究领域称为“计算复杂性”
- 事实：有些问题，已经找到了复杂性很低的算法；有些问题，虽然已经做了很多努力，也没有找到复杂性较低的算法

  没找到并不说明没有，也可能有但还没发现（例：素数性判断）
- 有些问题，已经证明了其最有效算法的复杂性

  有些问题，已经找到最有效的算法（改进余地不大，常量因子）

问题和复杂性

- 另一些情况：
  - 有些问题，已经证明求解它的最高效算法也一定具有指数以上的复杂性。这种问题称为“难解问题”
  - 只可能能有指数复杂性的算法，说明这些问题的规模较大的实例，（从实际的角度看）“不能”用计算机求解
- 注意上面“不能”的意义
  - 不是数学意义（或理论意义）上的“不能”。数学或理论的“不能”就是“不可计算”，可计算问题就是可求解的问题
  - 实际意义上的“不能”是说，虽然这些问题理论上可计算，但对于它的具有一定规模的实例，没办法实际地得到计算结果
- 对前面问题（10000次基本运算/秒，规模 100，$O(2^n)$ 算法）
  - 假如机器速度提高 1 万亿倍（10^{12}），约 $4.0*10^6$ 年能得到结果
  - 但规模 140 的问题还是需要 $4.0*10^{18}$ 年（可解规模提高得很慢）

计算复杂性

- 上面讨论区分了两种情况
  - 理论上不可计算的问题（复杂/过于一般，不可能实现计算过程）
  - 实际意义上不能计算的问题（复杂性高，不存在高效算法）
- 把计算复杂性纳入考虑，出现了另一个重要理论问题：

  是否存在本质上比图灵机更快的计算模型？

  即是问：是否存在“实际可计算能力”优于图灵机计算模型，它能有效解决某些在图灵机模型上不存在高效算法的问题？
- 人们提出了一个“扩展的图灵论题”：在任何（非并行）计算模型上多项式时间可解的问题，必然在图灵机上多项式时间可解
- 与图灵论题类似，这一问题无法证明，只能否证。但有如下事实

  已证明，目前已提出的各种重要计算模型，在多项式时间可解性方面与图灵机等价（也就是说，图灵机上需要指数以上时间求解的问题，在这些模型上也不存在多项式算法）

计算复杂性

- 根据计算复杂性，可以对问题做一种分层（复杂性分层）
  - 显然，属于低复杂性类的问题也属于更高的复杂性类
- 这方面最重要成果是确定了一大批有重要实际意义的问题，它们
  - 在具有无穷并行能力的计算机器上求解，时间复杂性为多项式的
  - 在只有顺序处理能力或有限并行能力的计算机器上，其复杂性至今未确定（找到的算法都为指数时间，但尚未证明只有指数算法）
- 由此引出的重要理论问题是：**P =?= NP** （**P** 与 **NP** 问题），问题涉及到两个复杂性问题类
  - 在顺序计算机器上多项式时间可解的问题类 **P**
  - 在具有无穷并行能力的计算机器上多项式时间可解的问题类 **NP**
- 美国克雷（**Clay**）数学研究所**2000**年**5**月宣布了七个悬赏一百万美元的“世纪数学难题”：**P-NP** 问题、霍奇猜想、庞加莱猜想、黎曼假设、杨-米尔斯理论、纳卫尔-斯托可方程、**BSD**猜想

计算复杂性

- P-NP 问题被列为第一数学问题（传统的重大数学问题均列其后），这一情况很有趣，说明了数学界对“计算”的重要性的评价
  - 原本人们认为计算太简单，是最简单的数学
  - P-NP 问题说明计算中存在极深刻的理论，其价值不亚于任何传统数学领域的理论。它有许多有价值的推论，人们已在用各种数学工具去研究它，解决它的过程必定给数学带来许多收获
- 这一问题还有极其重大的实际意义
  - 目前社会中几乎所有现代意义下的安全性，都依赖于 NP ≠ P。一旦有人证明了 NP = P，所有重要领域的许多系统（军事、金融、通讯等）都需要重建。同时我们也看到了计算能力的新边界，社会生产生活各方面的效率都可能大大提高
  - 如果证明了 NP ≠ P，我们的生活环境不会有什么变化，但人类又进一步认识到自己能力的局限性
- 从来没有过一个数学问题如此深刻地影响（威胁？）整个人类社会

NP 完全性问题

- 深刻的数学理论问题常显示出美妙的结构。P 与 NP 问题是这方面的典范。其中最有趣的问题之一是 NP 完全性
- NP 问题类里有一个子集称为 NP 完全性问题（类），有如下结论
  - 任意两个 NP 完全性问题可通过多项式时间复杂性的变换相互转换。因此只需多项式时间就可以基于一个 NP 完全性问题的解得到另一个 NP 完全性问题的解：只需定义两个适当的多项式时间的翻译算法，就能解决另一 NP 完全性问题
  - 任一 NP 问题都可通过多项式时间变换到某 NP 完全性问题
- 推论：任一个 NP 问题都可以通过多项式时间变换归结到某个 NP 完全性问题（也就是说，变换到任意一个 NP 完全性问题）
- 重要结论：
  - 要证明 NP ≠ P，只需证明一个 NP 问题没有多项式时间算法
  - 要证明 NP = P，只需证明一个 NP 完全性问题有多项式时间算法

几个 NP 完全性问题

现在举几个典型的 NP 完全性问题

- 货郎担问题：

  设有 n 个村庄，村庄之间均有路相连，每条路有一定长度

  要求找一条最短的路径，它经过每个村庄一次，最后回到出发点

- 货物打包问题：

  设有多种货物，每种有一定重量。货车一台有确定载重量

  要求选出一组货品，货车可以承载且在所有可能组合中重量最大

- 哈密尔顿圈问题：

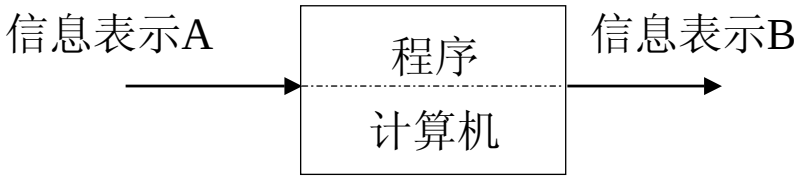
  在一个图中有一些点，有的点之间有连线

  问图中是否存在一条环路，它经过每个点且只经过一次

- 有关计算理论讨论至此暂告结束，网上可以找到许多参考材料

计算和数据表示

用计算机解决问题，可以认为是实现某种信息表示形式的转换



- 如果：
  - “信息表示A”表达了需要求解的某个问题
  - “信息表示B”表达了相应的求解结果
  - 可以认为：这个程序完成该问题的求解
- 为处理与问题有关的信息，必须用某种方式表示它，并将相应表示存入计算机。这种信息表示就称为（计算机处理的）数据
- 为有效使用，必须以某种方式把程序使用的数据组织好。程序越复杂，需要处理的数据情况越复杂，数据的良好组织就越重要

数据结构

- 数据（**Data**）：计算机程序能处理的符号形式的总和

  信息（**information**）是一个含义更广泛的概念

  一种说法：数据是编码的信息（信息的编码表示）

- 数据元素（**Data Element**）

  数据的基本单位，在程序中通常作为一个整体考虑和处理

- 数据结构（**Data Structures**）

  一组数据元素（结点）按照一定方式构成的复合数据形式

  以及作用于这些元素或者结构上的一些函数或操作

- 本课程将讨论一批典型的常用数据结构：

  表/堆栈/队列/链接表/字符串/树/二叉树/字典/集合/图

  帮助大家理解复杂数据的表示和处理技术，各种结构的性质（支持哪些操作，操作的代价等），进一步理解计算的本质

数据结构

- 一种数据结构是采用一套特定方式建立起来的一种数据组织结构（以数据元素为出发点），其特征包括几个方面（层次）：
  - 逻辑结构：数据元素之间有某种特定的逻辑关系。这是元素之间的抽象关系，与实现无关。抽象看，一个数据结构是一个二元组

    B = (E, R)

    E 是一些种类的数据的集合，**B** 的元素取自集合 **E**

    元素之间有关系 **R**，不同数据结构的元素之间的关系不同
  - 物理结构：数据的逻辑结构在计算机存储器中的映射（或表示），又称存储结构，或数据结构的存储表示
  - 行为特征：作用于数据结构上的各种运算。例如检索元素、插入元素、删除元素等一般性操作，还可能有一些特殊操作
- 具体实现问题：在编程语言里实现数据结构的具体方式和技术

  对 **Python**，还需理解其重要内置数据结构的实现和性质

Python 数据结构

- 典型的数据元素是不可进一步分割的原子，如

  整数、浮点数、布尔值

  每种编程语言都提供了一组基本数据类型， 如整数，浮点数，逻辑类型等。这些类型的数据元都应看作数据元素

  语言还为每个基本类型提供了一批操作

- 常规语言通常提供几种基本的数据组织机制
  - 用于把一组简单（或复杂）的数据元素组织为一个整体，满足程序中管理、操作和传输的需要
  - 有些语言，如 **C** 语言，只提供了几个基本类型和两三种数据构造机制，所有复杂的数据结构都需要自己定义
  - 另一些语言（包括 **Python**）本身就提供了一批高级的数据类型，其中一些实际上是最有用的数据结构。如 **Python** 的 **list** 等

Python 数据结构

- 上学期介绍了 **Python** 的一些标准数据类型，其中一些实际上就是非常有用的数据结构：
  - 正文序列类型 **str**
  - 序列类型 **list** 和 **tuple**
  - 集合类型 **set** 和 **frozenset**
  - 映射类型 **dict**
- 本课程中还会反复使用它们
  - 标准库还提供了另一些数据结构定义，如 **deque** 等
  - 下面简单罗列一些情况，请大家自己回忆和复习
- 本课程后面讨论中在这方面要强调的重点是：
  - 剖析这些数据结构的实现和性质
  - 帮助理解在写程序时应该如何使用它们，以及为什么

Python 数据结构（简单回顾）

- 字符串，类型名 **str**，不变（**immutable**）正文序列类型
  - 对象：字符的有穷序列
  - 访问操作：求长度，取成员，取子串（切片），查找子串，判断成员字符类型，等等
  - 基于已有字符串构造新串：改变大小写，拼接，格式化（变换格式），子串替换，切分，等等。都是构造新串
- 元组，类型名 **tuple**，不变序列类型
  - 对象：任意元素的有穷序列，一个元组的成员可为不同类型的对象
  - 序列的共有操作：成员判断（**in, not in**），取元素/切片，长度，元素检索/计数，最大/最小，拼接/重复拼接（产生新序列），等
  - 不变序列的唯一特殊操作是 **hash**，它从任意个不变序列生成一个整数，具体映射方式由系统的实现确定
  - 不变序列可以作为字典的关键码，可作为 **set/frozenset** 的元素

Python 数据结构

- 表，类型名 **list**，可变（**mutable**）的序列类型
  - 对象：任意元素的序列
  - 支持所有共有的序列操作
  - 所有可变序列操作（改变被操作的表）：元素替换（赋值），切片替换/删除，元素插入/删除，等等
- 集合，类型名 **set** 和 **frozenset**，分别为可变/不变汇集类型。其元素可以是数值或逻辑对象/字符串/不变序列等，元素值唯一
  - 对象：任意满足条件的元素的汇集
  - **frozenset** 支持所有不变集合操作：元素个数，元素判断，集合之间的关系（包含/相交），集合运算（并/交/差/对称差）等
  - **set** 还支持可变集合操作：修改集合的运算（并/交/差/对称差），加入/删除元素等
  - 集合的元素只能是不变对象，可求出 **hash** 值

Python 数据结构

- 字典，类型名 **dict**，可以看作关键码和值的二元组的集合，其中关键码必须是不变对象
  - 对象：一组关键码到一组值的映射
  - 最基本操作是关键码-值二元组的加入和关键码检索
  - 其他操作：元素个数，关键码存在判断，元素删除（基于关键码），另一些修改字典的操作，等
- 在 **Python** 的标准库里还定义了另一些数据结构
  - 后面讨论中可能介绍一些相关情况

抽象数据类型

- 假设要在程序里处理一些有理数数据。可能的方式：
- 直接用一对 **int** 变量表示一个有理数

  不好！很难用，操作起来很不方便

  难以维护有理数数据的完整性（特别是程序里有多个有理数时）
- 如上学期所示，用包含两个整数成员的元组表示有理数
  - 为有理数定义一批操作，总用它们处理表示有理数的元组对象
  - 在任何使用有理数的地方都不直接操作这种元组对象的成分
- 也不够好！一些问题：
  - 用元组表示的有理数不是类型，没有类型名，不像已有 **Python** 类型那样好用，相关操作与对象没有清晰的关联
  - 无法与其他元组相互区分，如完全可能用整数成员的元组表示其他数据，例如坐标平面上整点的坐标，这时 **(2, 0)** 是合法数据

抽象数据类型

- 抽象数据类型是一种组织复杂程序的重要思想

  基本想法来源于人对复杂现实世界的处理方法和技术

- 在思考/讨论/处理现实世界复杂问题时，人们依靠一些重要的抽象概念
  - 考虑金融问题时，重要的概念包括股票/债券等
  - 讨论数值变化关系时，定义了函数/参数/导数等概念
- 一个概念通常有些相关的数据（属性）和操作，以债券为例
  - 数据属性如价格/利率/到期日等
  - 操作如设定出售价格，计算到期收益等
- 如果需要在计算机系统里处理这些概念，就需要反映上面的认识
  - 用程序里的概念反映真实世界中的概念，在其中组合起与有关概念相关的属性和操作
  - 抽象数据类型就是基于数据抽象，组织复杂程序的一套思想

抽象数据类型

- 抽象数据类型的思想：
  - 采用某种数据形式表示所需的数据“类型”
  - 并定义一组操作，实现对有关数据对象的所需操作
- 用元组实现有理数（如上学期的示例），可以看作抽象数据类型的一种实现。但这种实现不够理想，因为它不抽象

  只是元组的具体应用，不是新类型，不具有类型的特点

- **Python** 内置类型是“抽象数据类型”，符合上面的思想

  许多 **Python** 标准库包实现某种数据类型，它们像内置类型一样是实实在在的抽象，如标准库手册第 **8** 节介绍的各种数据类型包

- 在写复杂程序时，如果能自己创建与内置类型性质和使用方式类似的“用户定义类型”，有可能把程序组织得更好

  这样定义类型是建立“数据抽象”（与 **[**计算**]** 过程抽象对应），定义的一个抽象数据类型成为程序的一个组件

抽象数据类型和 Python 的 class

- Python 提供了支持定义数据抽象的机制
  - 定义数据类型的标准 Python 机制是 class 定义（类定义）

    定义一个类（class）就是定义一个新类型
  - 定义好的类可以像内置类型一样使用，包括生成类的实例（类的实例称为对象，object，在前面讨论中反复提到）
  - 在类定义里，可以为本类的实例定义一组相关操作

    对一个类的实例对象，可以使用类里定义的这些操作
- Python 的基本系统就是基于类和对象构造起来的
  - class 是 Python 语言最基本的概念，许多重要问题需要用 class 及其性质解释

    例如：Python 内部类型的性质，异常和异常处理等
  - 标准库包的抽象数据类型都是用 class 定义的

类与面向对象的编程

- 定义类，生成类的对象，基于这种对象组织和描述计算，这一套做法称为面向对象编程（Object Oriented Programming，OOP）
  - OOP 是软件领域最重要的技术，有利于分解系统的复杂性
  - 许多复杂软件系统都是用 OOP 技术开发的
  - Python 也称为是一种面向对象的编程语言
- 类是在基本计算结构（表达式/语句/函数定义）之上的高级结构

  语法：

  ```
  class C :  # 定义一个以 C 为名字的类
      语句块
  ```

  或

  ```
  class C (B) : # 基于已有类 B 扩充定义一个新类 C
      语句块
  ```

类定义实例：有理数类

- 类定义中的程序块里通常是一组函数定义
  - 用类定义数据抽象时，这种函数定义描述了该类的对象的行为
  - 可以定义函数去创建、操作、生成该类的对象
- 下面考虑定义一个有理数类
  - 给这个类取名 **rational**
  - 该类的对象是具体的有理数
  - 需定义各种有理数运算（操作），以便在程序里方便地使用它们
- **rational** 类的定义：

```
class rational :
    __init__(self, num, den) : # __init__ 函数创建类的对象
        self.num = num          # 第一个参数 self 表示被创建对象
        self.den = den          # 通过 self 和圆点给对象的成分赋值
        ... ...                 # 成分的名字根据需要选择
```


类定义实例：有理数类

- 名字为 __init__ 的函数称为它所在类的构造函数
  - 创建这个类的对象时，自动调用这个函数
  - 在创建类的对象时，不需要为 __init__ 的第一个参数 **self** 提供值，但需要为其他参数提供值

```
r = rational(2,3)  # 建立一个 rational 类对象，2 和 3 是其成分
```

- 定义一个操作输出有理数对象
  - 设计输出，采用 分子/分母 的形式，函数取名 **print**
  - 函数定义（放在类定义的程序块里）

```
def print(self) :
    print(self.num, "/", self.den)
```

  - 函数的使用

```
r.print()
```

类定义实例：有理数类

- 为类的对象定义操作（称为方法）和使用的要点：
  - 方法的定义写在类的程序块里
  - 方法的第一个参数表示操作对象，通常用 **self** 作为参数名，方法体里用该参数加圆点的形式引用对象的成分（取值或赋值）
  - 当变量的值是本类的对象时，用圆点加方法名的形式调用方法，并为方法（除第一个参数外）的其他参数提供实参
- 定义一个到 **str** 类型的转换函数：

```
def __str__(self) :
    return str(self.num) + "/" + str(self.den)
```

人们在定义类时，经常为其定义一个到 **str** 的转换函数

主要用途是从对象生成字符串，以便输出或在其他地方使用

类定义实例：有理数类

- 有理数加法，例如用 **plus** 作为方法名：

```
def plus(self, another) :
    den = self.den * another.den
    num = (self.num * another.den + self.den * another.num)
    return rational(num, den)
```

注意：这里调用 **rational** 构造新有理数对象，而不直接构造 **tuple**

下面会看到这种统一描述带来的收益

- 使用：

```
r1 = r.plus(rational(3, 4))
r2 = r1.plus(r)
```

- 我们可能觉得用这种形式写计算不自然，最好能用内置数类型的运算的写法，用 **+ - * /** 等运算符

在 **Python** 里可以做这件事，需要用一些特殊的方法名

类定义实例：有理数类

- 表示运算符的特殊名见语言手册 **3.3.7** 节，包括：

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__pow__(self, other[, modulo])
... ...
```

- 模拟算术运算的加法方法定义：

```
def __add__(self, another) :
    den = self.den * another.den
    num = (self.num * another.den + self.den * another.num)
    return rational(num, den)

r3 = r2 + r1
```


类定义实例：有理数类

- 使用这个 **rational** 类建立对象，做一些运算

```
>>> x = rational(3,5)
>>> x = x + rational(7,10)
x.print()
65 / 50
```

易见，定义的类没做分数化简，会导致分子分母变得很大

- 化简的数学基础很清楚，但怎么修改程序？
- 可以修改实现加法的方法，加入与化简有关的语句

  但还有减法、乘法、除法等。这样需要做许多重复工作

- 另一可能性是修改构造函数

  如果所有有理数都是用构造函数建立，一个修改就能解决所有问题

类定义实例：有理数类

- 考虑所有可能情况（如分子/分母是负数，错误值等），有下面定义：

```
def __init__(self, num, den = 1) :
    if type(num) != int or type(den) != int :
        raise TypeError
    if den == 0 :
        raise ValueError
    sign = 1
    if (num < 0) :
        num, sign = -num, -sign
    if (den < 0) :
        den, sign = -den, -sign
    g = gcd(num, den)
    self.num = sign * (num//g)
    self.den = den//g
```

这里还需要一个求最大公约数的函数，可以是全局定义的，也可以定义在有理数类的里面

类定义实例：有理数类

- 增加其他运算已经没有任何困难了，请自己补充完整
  - 剩下的问题一方面是数学，大家都熟悉
  - 另一方面是具体运算的编程，可以参考 __add__ 的实现
- 把 rational 类的完整定义放入一个文件，作为一个模块
  - 如果需要用有理数，import 这个模块，类型 rational 就有了定义
  - 可以用 rational 建立有理数对象，用它提供的功能操作有理数
  - 从使用的角度看，这个 rational 与内置的 int, list 等类型没有差别
- 对于 class 定义，还有一些功能和规定
  - 有关情况可查看语言手册，其他参考书
  - 上面讨论的是最规范的使用，基本满足本课程的需要
  - 后面会根据情况考虑

注意看课程主页的作业（下午布置）

下次课：10月9日

上机：第3周9月30日晚停一次，10月7日属于放假期间，
实际上机从第5周开始

答疑定在周四下午，理科一号楼1480（我的办公室）

问题？