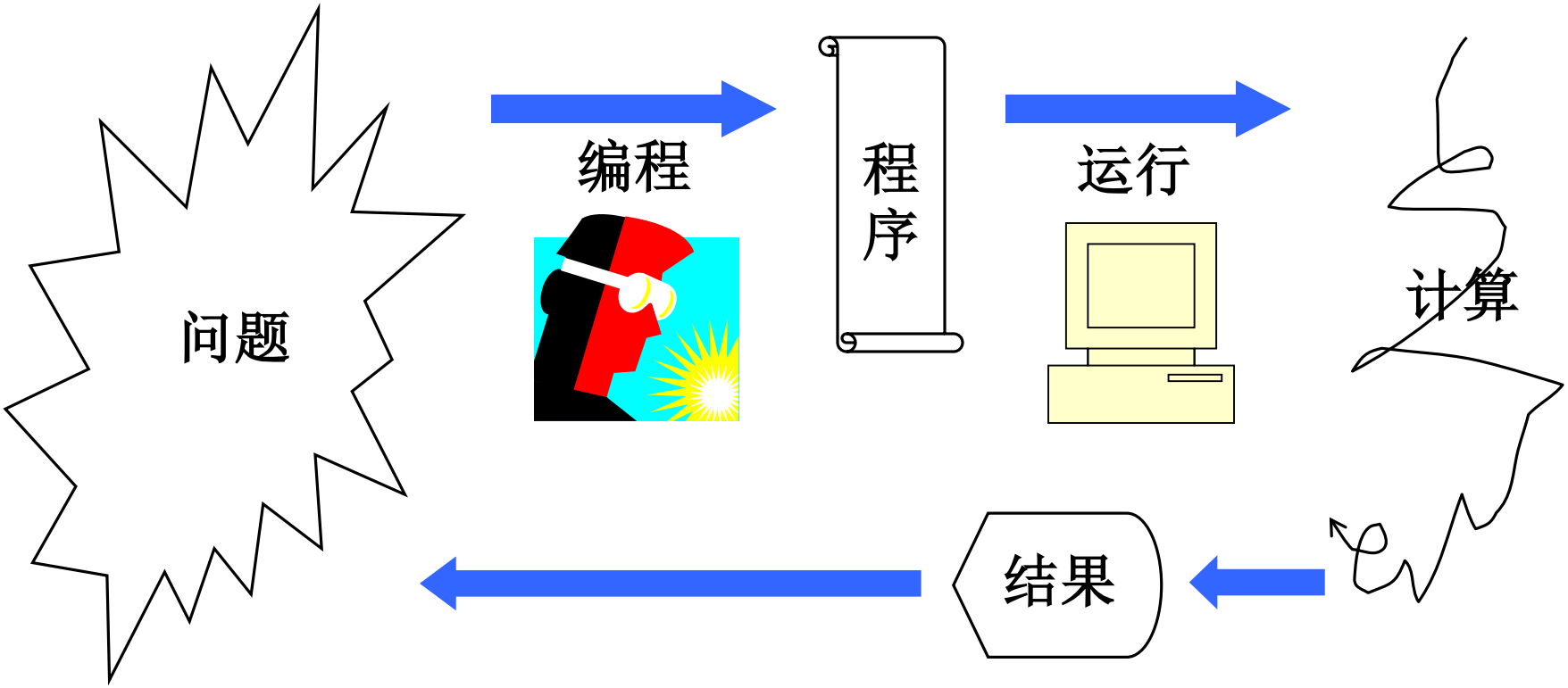
6.问题和程序

- 从问题到程序
- 编程实例
- 递归程序
  - **Fibonacci** 序列，最大公约数
  - 相互递归
  - 容易用递归解决的问题
- 给程序计时
- 定义函数

问题和程序

问题、程序和计算



问题和程序

- 编程序是为了解决问题。什么是问题？

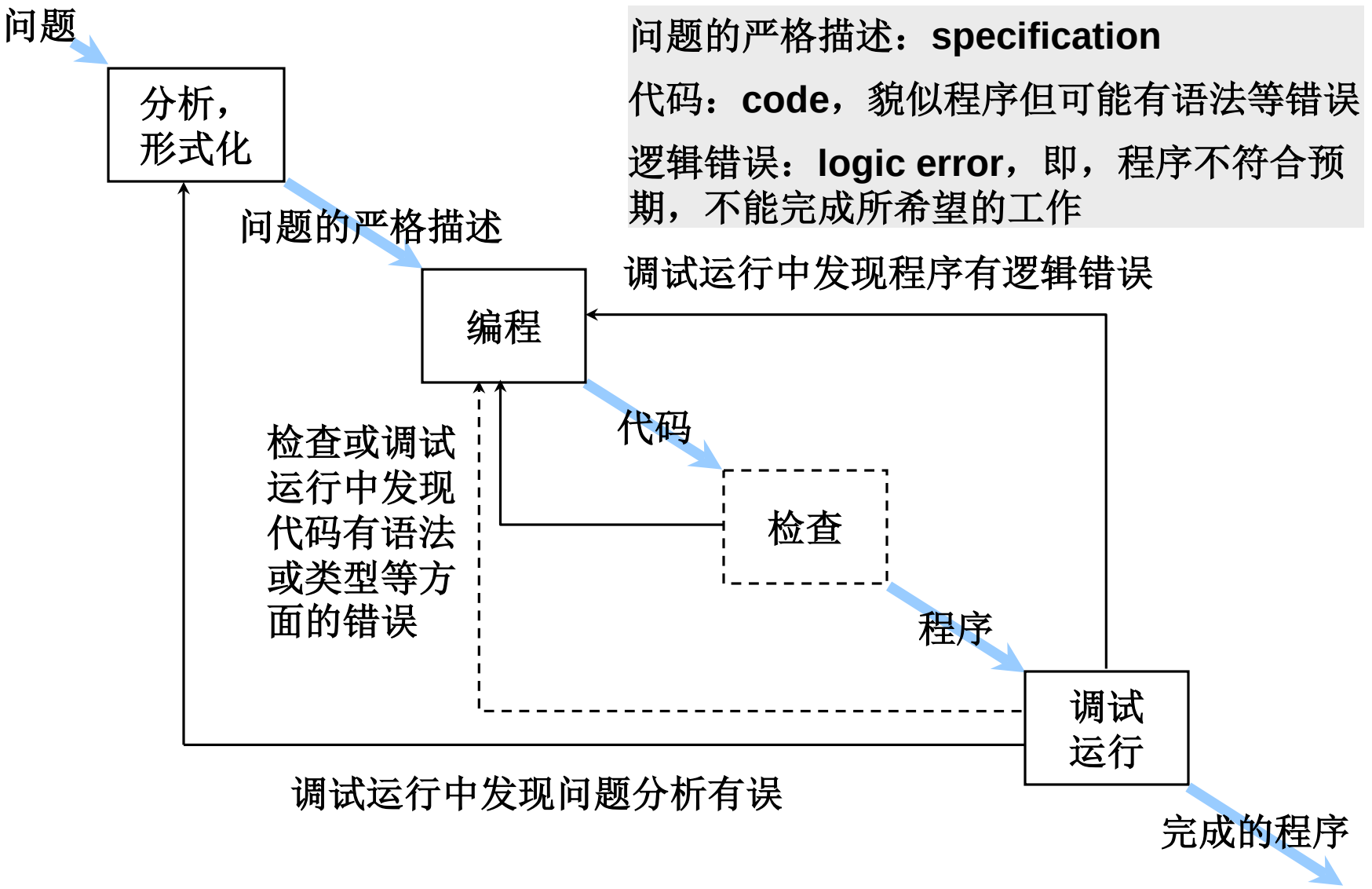
  直观的，如一道习题，…，直至一项实际需要完成的工作

- 问题是一种客观的需要完成的任务
  - 首先需要考虑该“问题”能否用计算机解决，或者是否可能从中抽取出一个或几个能用计算机解决的问题
  - 能用计算机解决的问题，理论有边界，实际中不清楚。研究和开发正在不断扩展实际应用领域
  - 下面只考虑可能用计算机解决的问题
- 一个程序是
  - 对一个问题的求解过程的精确描述
  - 用一种计算机能处理的语言表述

问题和程序

- 一般说，“问题”和“程序”距离很远：
  - 问题的自然描述通常采用自然语言，常以“是什么”的方式表达。不精确，非形式化，是描述性的
  - 程序用精确的编程语言描述，说的是“怎么做”
- 用计算机解决问题的过程：
  - 设法得到（写出）问题的描述（非形式的）
  - 将问题严格化（仍然为“是什么”，但更严格准确）
  - 设计问题的解决过程（怎样做？），直至得到“算法”
  - 用编程语言写出程序
  - 设法确认写出的程序解决了问题。可能反复

从问题到程序



问题的严格描述：**specification**

代码：**code**，貌似程序但可能有语法等错误

逻辑错误：**logic error**，即，程序不符合预期，不能完成所希望的工作

例：检查整数是否完全平方

- 整数 n 是完全平方，如果存在整数 m 使得 m*m = n

  这是“是什么”，是严格定义

  判断 n 是否完全平方，是一个问题

- 计算问题：试探能否找到一个 m 使 m*m = n

  怎么找？

- 解决问题的简单方法（算法的思路）：

  通过检查一些整数找到所需的 m，或确定其不存在

  这是一种常用计算模式：在一组候选里找出答案

  当然可以考虑其他方法

- 要保证：如果存在这样的 m，就一定能找到

  为此，只需保证：所检查的整数集合覆盖可能的 m

例：检查一个整数是否完全平方

- 精确化（向着计算的方向。每步都可能有许多选择）
  - 怎么选择要检查的整数？
  - 例如：顺序检查，为此要确定检查的范围
  - 其他可能性？
- 假设确定采用顺序检查一个整数区间的方法
  - 确定检查的整数区间

    即，从哪里开始，到哪里结束
  - 要保证区间足够大，不漏掉可能的 m
- 简单选择：m 从 0 开始，直至 n

  不难发现，检查到 m*m 不小于 n 就足够了（数学）

例：检查一个整数是否完全平方

- 从确定的区间里一个个取数检查
  - 反复检查是重复计算
    - 应该用循环表示
    - 区间确定循环变量的取值范围
  - 写出程序已经很自然了
- 应该定义为一个函数，是个判断函数，也称谓词（predicate）
- 实现的程序对不对
  - 通常是通过测试检验
  - 但，做多少测试才足够充分？
- 最后一定要问：这个程序好不好？有改进的可能吗？

  请自己考虑改进

例：生成并输出 1 到 200 的完全平方数

- 一种方法：检查从 1 到 200 的整数，遇到完全平方就输出
- 变量和循环：
  - 需要有一个变量，它的值遍历 1 到 200 的整数
  - 循环每迭代一次，该变量加一
  - 值超过 200 就结束
  - 可以直接用 for 的循环变量作为遍历整数的变量
- 代码梗概

```
for k in range(1, 201) :
    if k 是完全平方 :
        输出 k
```

例：生成 1 到 200 的完全平方数

- 另一方法：

  要输出的也就是从 1 到某数的平方，要求平方不超过 200

- 变量和循环
  - 用一个变量遍历从 1 到某数的区间，输出各个数的平方
  - 当它的平方大于 200 就结束
  - 用 for 还是用 while ?
- 代码的梗概：

```
k = 1
while k * k <= 200 :
    输出 k * k
    k += 1
```

例：生成 1 到 200 的完全平方数

- 另一方法，基于公式

$$(n+1)^2 = n^2 + 2n + 1$$

- 基于这个公式
  - 1 的平方已知
  - 从 1 的平方可以算出 2 的平方
  - ……
- 考虑：
  - 需要一个变量 s 表示平方，一个变量表示当前的 n
  - 每次迭代更新 s 到下一个平方，n 加一
  - 循环中的不变关系：s 总是 n 的平方

例：判断素数

- 素数是重要的数学概念
  - 没有除 1 和其自身以外的因子（没有真因子）的自然数
  - n 的因子：能整除 n 的正整数
- 整除，除的余数为 0

  可以用表达式 n % k == 0 描述
- 朴素的素数判断方法：
  - 用从 2 开始的整数去检查是否因子
  - 直至试的足够多，能确定 n 不可能有真因子为止
- 用循环描述检查过程，到什么时候结束？

  如果 n 有真因子，一定有小于等于其平方根的真因子

例：判断素数

- 定义函数（谓词）**is_prime**
- 对于 **100** 位的整数，判断是否素数要检查 10^{50} 个数，太耗时
- 有关数论的研究有许多与素数有关的成果
  - 其中一些成果可能用于素数判断
  - 人们提出了一些不同的算法
  - 在网上可以搜索到许多情况

    可以找到一些有趣的算法，可以自己实现试试
  - 最近一大进展是 **2002** 年印度科学家提出的 **AKS** 算法
- 素数判断在许多计算领域有重要应用

  特别是密码和安全领域，是今天计算机安全的基础

递归函数

- 递归函数在其定义里调用自身，进一步介绍一些情况
- 递归定义的特点：
  - 基础情况，可以简单地直接得到（计算出）结果
  - 一般情况：把计算归结到同一问题的更简单情况
- 例：求幂函数（指数是自然数）

  一个递归定义：

$$x^n = \begin{cases} 1 & \text{when } n = 0 \\ x \times x^{n-1} & \text{when } n > 0 \end{cases}$$

- 求幂需要做多少次乘法？
  - 这里求 n 次幂，需要做 n 次乘法
  - 能不能少做一些？

乘幂函数

- 该注意到一些特殊情况和例子

  例如：$2^8 = ((2^2)^2)^2$

  平方就是自乘，一次乘法

  这种性质有可能用来减少乘法的次数

- 通过分析，把上面这样的特殊情况推广到一般，可以得到求幂的另一个递归定义：

$$x^n = \begin{cases} 1 & \text{when } n = 0, \\ x \times x^{n-1} & \text{when } n \text{ is odd}, \\ (x \times x)^{n/2} & \text{when } n \neq 0 \text{ is even}. \end{cases}$$

- 同一个问题可能非常不同的算法，算法的性质非常不同
- 下面考虑另一个例子

Fibonacci 序列

- **Fibonacci**（斐波那契）序列是一个重要的整数序列

  因意大利数学家斐波那契**(1175-1250)**命名，最早源于印度

  有大量数学结果，在计算领域有许多特别重要的应用

- **Fibonacci** 序列的定义
  - $F_0 = 0, F_1 = 1$ （基础情况）
  - $F_n = F_{n-1} + F_{n-2}$, 对所有 $n > 1$（一般情况）

  序列的前几项：**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...**

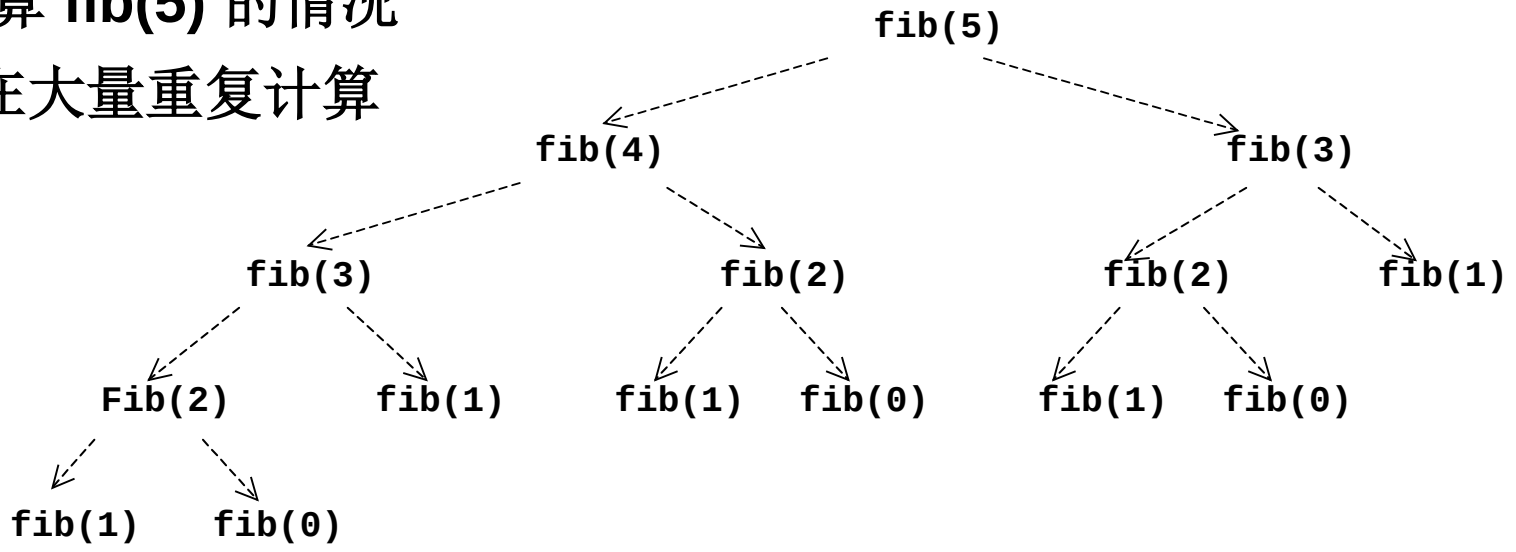
- **Fibonacci** 序列的定义是递归定义
  - 基础情况，一般情况可归结到更简单情况（下标更小）
  - 可以直接做出一个递归定义的函数
- 函数定义和计算试验

计算的时间

- 现在的计算机非常快，但计算也要花时间
  - 有些程序的计算可能花很长时间
  - 解决某些复杂问题的程序必然需要非常长的时间
- 函数 **fib** 的定义很好，与数学定义直接对应

  但也不好：计算要花费很长时间
- 看计算 **fib(5)** 的情况

  存在大量重复计算



计算有代价

- 实际计算由现实世界中的设备（目前是电子计算机）完成
  - 不是数学，不是抽象的理论
  - 每步计算都有代价，即使非常小但也不为 0，累积起来就可能超过给定的任意大的数（这是数学）
- 有代价是计算的一种本质特征，相关理论问题后面讨论
- 一些情况：
  - 可能存在从理论上看能用计算机解决的问题，但我们不可能等到计算机给出解的那个时间
  - 编程时需要考虑计算的时间，寻找解决问题的高效方法（算法）永远有意义
  - 有些实际问题对计算完成的时间非常敏感

    例：ABS/EBD(电子制动分配)系统，天气预报系统等

再看 Fibonacci 序列

- 对 Fibonacci 数的计算，换一种看法：

  基本情况已直接给定，由 F_{n-1} 和 F_n 可以递推得到 F_{n+1}

- 现在考虑基于一个递推循环的函数，从 F_0 和 F_1 出发，一步步递推，直至得到所需的 F_n

- 考虑循环中的变量安排：
  - 用 f1 和 f2 记录已知的两个相邻 Fib 值
  - f1 + f2 是下一个 Fib 值
  - 需要知道算到第几个 Fib 值，k 记录变量 f1 的值是 F_k
  - f1, f2 和 k 需要在每次循环中更新

```
f1, f2 =  f2, f1+f2
k += 1
```

也可以用递归函数描述这种计算

存在更快的算法

求最大公约数

- 定义：两个数 m 和 n 的最大公约数，就是既能整除 m 又能整除 n 的最大的正整数
- 最简单而直接的想法是用一系列的正整数试验
- 一种考虑
  - 从 1 开始试，1 总可以整除 m 和 n
  - 递增，发现同时为 m 和 n 约数时记录它
  - 试到超过 m 和 n 中较小的一个，就可以结束
  - 记录的最大的那个约数，就是 m 和 n 的最大公约数
- 假定 m >= 0, n > 0，容易写出函数定义

  如果允许参数是任何整数（包括负数），需要加一些前处理

求最大公约数：辗转相除

- 还可以从大到小循环检查

  找到的第一个能同时整除两个数的 **d** 就是最大公约数

- 求最大公约数的经典算法是辗转相除法，又称为欧几里得算法。它利用下面关系

$$\gcd(m,n)=\begin{cases} n & m \bmod n = 0 \\ \gcd(n, m \bmod n) & m \bmod n \neq 0 \end{cases}$$

- 利用这个关系实现一个求 **gcd** 的函数
- 改造得到的函数，增加检查，可以能得到一个可以对任一对整数求最大公约数的函数

递归和循环

- 是不是递归函数一定比循环函数慢？

  为什么考虑递归程序有价值？

- 回答：
  - 不一定
  - 例如 **Fibonacci** 的递归和循环实现采用了不同算法，完全可以用递归实现第二种算法
- 为什么需要递归定义？
  - 不少算法用递归的方式描述简单而自然，正确性很容易看清楚。用循环方式描述却很难/复杂/意义不清晰
  - 后面学习中还会看到，在计算中需要处理的许多结构本身就是递归的，用递归的方式处理非常自然

递归实例：换硬币

- 一个递归程序实例

  也可以用循环的方式写程序，但比较复杂，也不那么清晰

- 人民币硬币有 1 分、2 分、5 分、10 分、50 分和 100 分。给定一定款额，问将其换成硬币有多少中不同的兑换方式
- 递归的看法：币值 n 的兑换方式数等于
  - 用了一个硬币 a 之后 n – a 的兑换方式数，加上
  - 不用硬币 a 时 n 的兑换方式数
- 上面是递归。有几种（基础）情况可以直接得到结果：
  - n 等于 0，说明找到一种兑换方式（1 种兑换方式）
  - n 小于 0，说明没找到兑换方式（0 种兑换方式）
  - 货币数等于 0 说明没找到兑换方式（0 种兑换方式）

递归实例：换硬币

- 考虑算法的实现，其中有些问题需要解决
- 为取得硬币的币值，把硬币排顺序后编号

  1 到 **6** 对应到 **1** 分到 **100** 分的硬币

  定义一个函数，实现这种对应。减一种硬币时编号范围减一
- 前面有关分析可以直接翻译为一个递归定义的函数

  ccoins

  定义一个主函数，调用 **ccoins(6, n)**
- 这个函数用递归的方式写，很简单
  - 函数定义直接对应于前面的分析
  - 也可以用循环的方式写出程序，但比较复杂，也不那么清晰。有编程经验的同学自己试试，需要用一些高级技术

相互递归

- 有时需要两个或多个函数相互调用
  - 例如在 f 里调用 g，在 g 里调用 f
  - 这样的函数称为相互递归的函数
- Python（只）要求函数在使用时必须有定义
  - 前面定义的函数可以调用后面定义的函数

    只要实际调用时被调用函数已经定义
  - 定义函数时，用到的函数可以还没有定义，但执行一个函数时（调用函数时），执行中用到的函数必须都有定义
- 举例：两个相互递归的函数

为什么定义函数？

- 函数就是实现某种计算的一段代码的抽象

  完全可以把这段代码直接写在调用这个函数的地方

  为什么要把它定义为一个函数？

- 例，可以定义函数 **cube** 计算立方，而后在需要时写调用

  c1 = cube(x)

  c2 = cube(y)

- 也可以不定义函数，在这些地方直接写

  c1 = x * x * x

  c2 = y * y * y

- 采用前一写法有什么好处？（功利性的问题）

为什么定义函数？

- 作用1：可能缩短程序（最直接的功利性原因）
  - 如果函数的定义体比较长，而且在程序里多处调用
  - 定义函数只写一次，调用代码很短
- 作用2：定义函数就是定义新的编程概念，扩充编程语言
  - 原来语言里没有立方的概念，定义 **cube** 函数就是给语言加入一个新概念，写/说/思考程序都能用这个概念
  - 无论写多少次 **x * x * x** 也没引进新概念
- 作用3：把一种有用的计算的定义集中到一个地方进行描述，发现错了可以在一个地方修正，也有利于今后的程序修改

  例如，后来可能发现新的方法完成这一计算

为什么定义函数？

- 作用4：功能分解，分解程序的复杂性
  - 把要开发的复杂功能分解为一些概念清晰，功能较为简单的待开发函数
  - 理清了程序的实现结构，分解了困难
- 作用5：函数作为开发的单位
  - 方便开发的分工
  - 依赖于局部开发、检查、调试、发现和更正错误
- 作用6：开发出来并经过仔细优化和检查的通用功能有可能重用。Python 的标准库和其他库是这方面的典型
- 还有很多可能的作用

学习定义函数

- 在学习基本编程的阶段，学会定义函数是非常重要的
  - 通过分析问题，确定应该把什么定义为函数
  - 学习基于抽象函数的思考方式
  - 学会描述函数的头部，以及写好函数体
- Python 函数定义的头部无法描述对参数的要求
  - 实际上，大多数函数对所需参数有特殊要求
  - 例如：
    - 要求某参数是整数，或数值，或是函数等（类型要求）
    - 要求某参数的值必须为正（值要求）
  - 可用文档串说明这种要求，也可以用条件检查或者断言贯彻实施这种要求，或者用条件语句检查这种要求

学习定义函数

- 考虑好定义一个函数（从概念上确定了该函数做什么）后，首先把函数头部设计好，函数头部是函数与外界的接口：
  - 给函数命名
  - 确定函数的参数，各自的作用和类型（Python 里写形参时无法描述类型，可以在函数体里通过检查的方式提出要求）
- 对一个函数的两种观点：
  - 内部观点（函数定义者）关心与函数实现有关的问题：采用什么算法？采用什么程序结构和语言机制？实际参数如何在计算中使用？如何得到结果（返回值）？等等
  - 外部观点（函数使用者）关心与函数使用有关的问题：这个函数能做什么？它需要哪些参数？各为什么类型？函数的结果是什么（什么类型的值）？等等