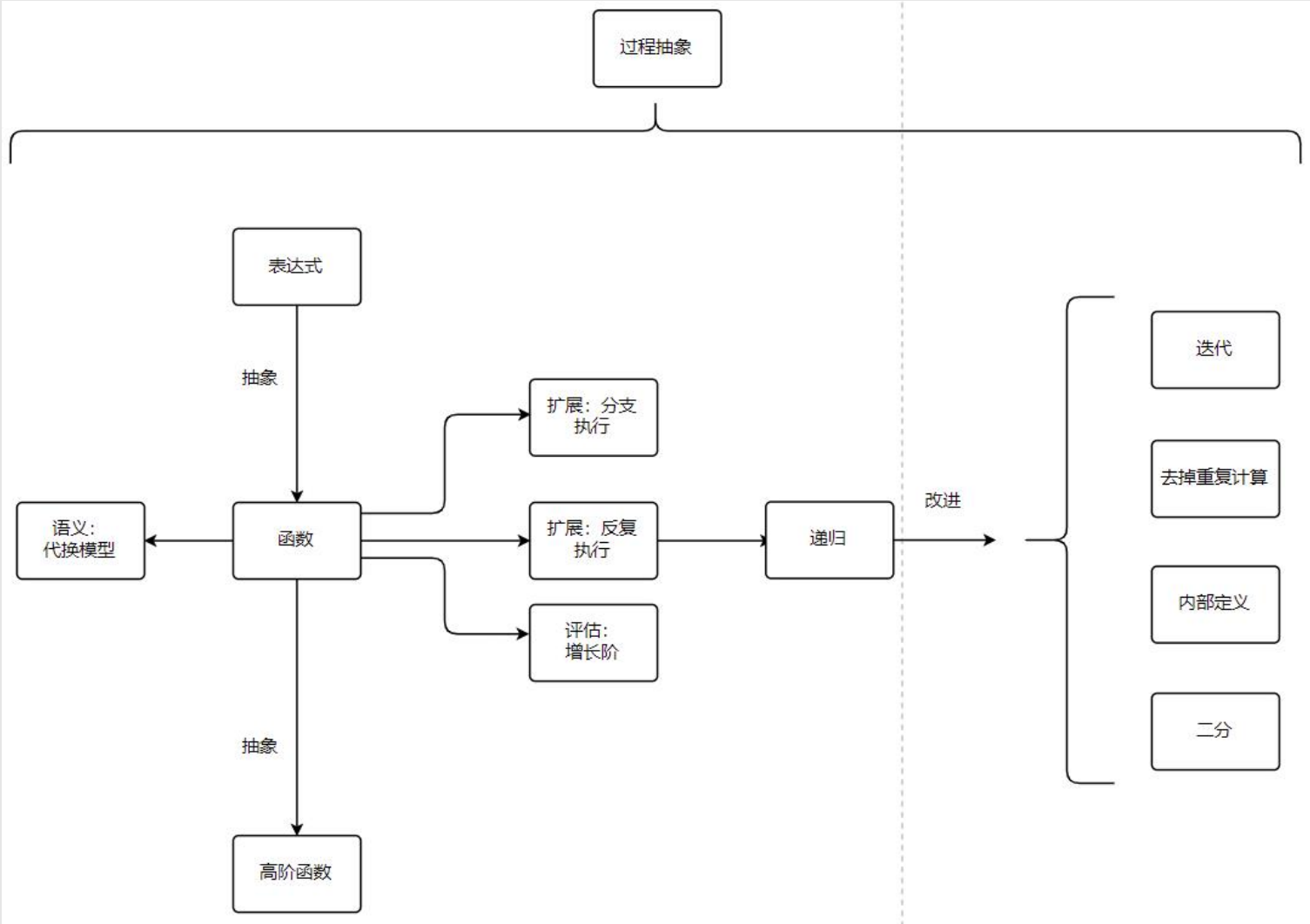
计算机科学导论-SICP
第一章 构造过程抽象
1.2小节 过程及其产生的计算

王超

Center for Research and Innovation in Software Engineering (RISE), Southwest University

2022 年 9 月 8 日

第一章知识间的关系


过程抽象
表达式
抽象
扩展：分支
执行
扩展：反复
执行
评估：
增长阶
语义：
代换模型
函数
递归
改进
迭代
去掉重复计算
内部定义
二分
抽象
高阶函数

1.2节讲了什么故事

- 尽管有些意外，我们目前所学的Scheme已经足够解决很多不平凡的数字计算问题
- 通过一种叫做递归的方式，可以非常精妙的使用过程完成计算。
- 递归是程序设计的核心思想之一，需要认真学习
- 1.2节以递归为核心，讲解了如何使用递归设计程序，程序的评估，以及对递归的改进

1.2节的PPT涉及哪些内容

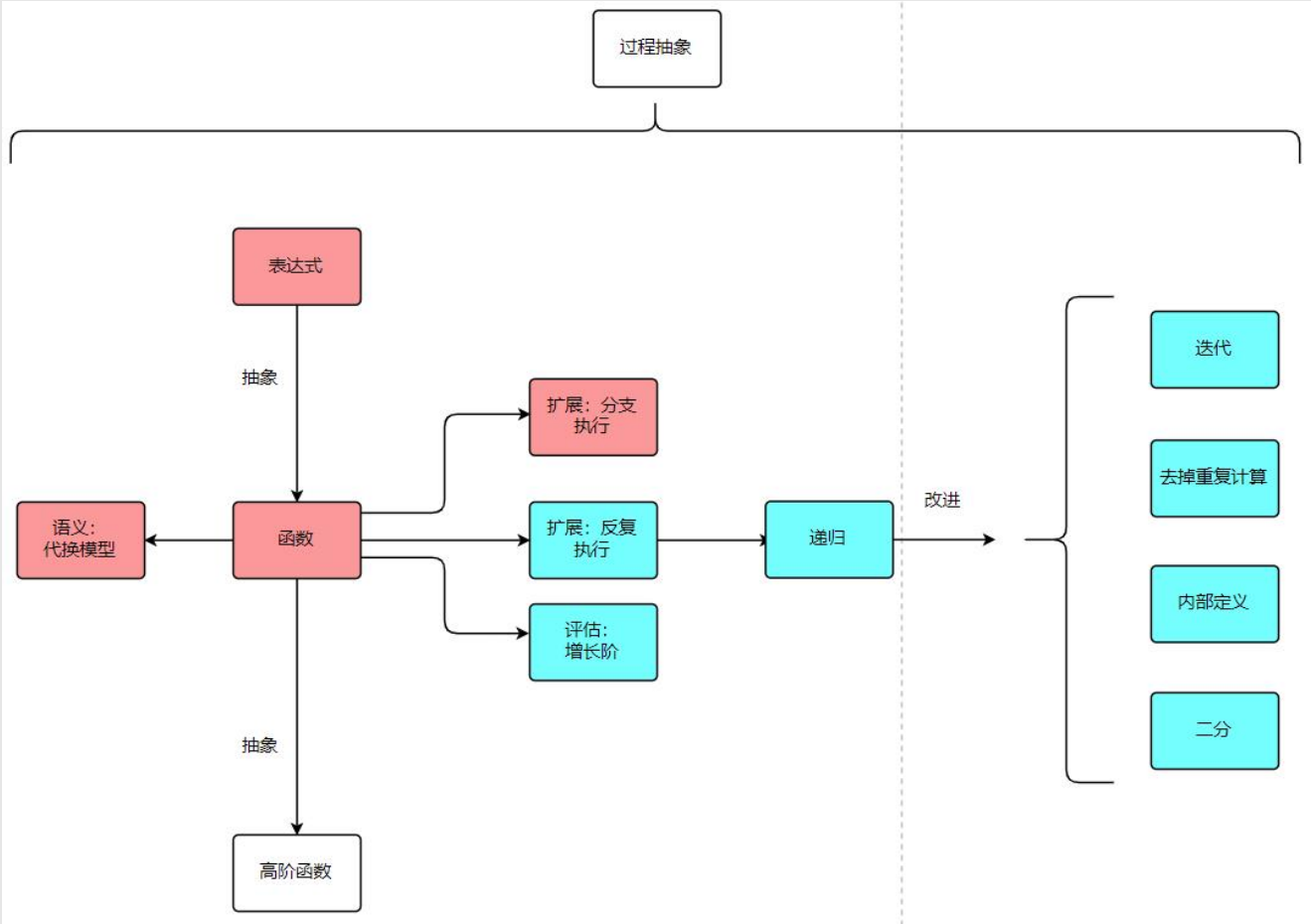
- 递归过程
- 优化：迭代的计算过程
- 优化：去掉重复计算
- 优化：二分
- 经典例子：斐波那契数列，通过牛顿法计算开平方，最大公约数，计算幂

内容调整

- 1.2.2节的找零钱算法不讲
- 1.2.2节递归的斐波那契数列算法的空间增长阶不讲
- 1.2.6节不讲

1.2节的内容

红色为已讲解内容，蓝色为本次要讲解的内容。



大纲

引入

完成一些任务需要反复计算，且次数不固定

- 例如，计算n的阶乘
- 计算6的阶乘，需要5次乘法：$6! = 6 * 5 * 4 * 3 * 2 * 1$
- 计算10的阶乘，需要9次乘法
- 如果过程(factorial n)计算n的阶乘，则过程需执行n-1次乘法
- 但是一个过程只能调用其他过程固定次。例如sum-of-squares调用两次square，square调用一次乘法
- 如此看来，一个过程似乎只能完成固定次数的基本运算
- 如何完成不固定次数的计算

令人惊奇的是，我们目前可以完成这个任务

初见递归

- 一个过程在其过程体内可以使用过程，包括使用自身
- 过程在其过程体内使用自身，这种现象叫做递归
- 初看起来令人迷惑，我们稍后讲解递归的原理

如何使用递归编写过程(factorial n)，完成n!的计算

- (factorial n)的值取决于n的值
- 当$n = 1$时，(factorial n)=1
- 当$n > 1$时，(factorial n)=n*(factorial n-1)，因为$n! = n * (n-1)!$
- Scheme可以很方便的把这两个等式写为代码

代码建议

- 写在上面的代码，在再次点击“运行”之后也不会消失
- 因此，建议把过程定义等会被反复使用的代码写在上面，把过程使用等程序员临时执行的代码写在下面

```
#lang sicp

(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (factorial 4)
24
>
```

(factorial 4)的求值过程

- (factorial 4)
- (if (= 4 1) 1 (* 4 (factorial (- 4 1))))
- (* 4 (factorial 3))，计算内层的(factorial 3)
- (* 4 (if (= 3 1) 1 (* 3 (factorial (- 3 1)))))
- (* 4 (* 3 (factorial 2)))，计算内层的(factorial 2)
- (* 4 (* 3 (if (= 2 1) 1 (* 2 (factorial (- 2 1))))))
- (* 4 (* 3 (* 2 (factorial 1))))，计算内层的(factorial 1)
- (* 4 (* 3 (* 2 (if (= 1 1) 1 (factorial (- 1 1))))))
- (* 4 (* 3 (* 2 1)))
- 24

- 递归算法的设计流程似乎是先写出一个等式，然后再把等式改写为代码

关于递归进一步的问题

- 这个等式需要满足什么条件。或者说，什么样的等式可以完成工作
- 如何构造这样的等式
- 什么样的递归算法是“好的”，什么样的递归算法是“坏的”

贸然使用自己定义自己容易导致循环论证

- 如果一个过程是通过“循环论证”定义的，那么其求值永不终止
- 例如，定义过程f为f(n)=f(n)+1，那么f(1)=f(1)+1=(f(1)+1)+1=...
- 递归定义不只是自己定义自己，还有一些额外的约束
- 递归定义不会引入循环论证

递归

递归过程(f x)

- f的定义包含递归等式和递归出口
- 递归出口：当x的值足够大（或足够小）到某个阈值时，(f x)可以直接得到计算结果
- 递归等式：等式左边是(f x)，等式右边只能出现所有已经定义的过程，以及过程f
- 如果参数足够小时f直接得到结果，则递归等式中等式右边f的参数一定要小于x。这样，随着程序执行，待求值的f的参数值不断变小
- 否则，如果参数足够大时f直接得到结果，则递归等式中等式右边f的参数一定要大于x。这样，随着程序执行，待求值的f的参数的值不断变大
- 等式右边每向内层求值一次f，都会导致参数更加接近阈值
- 因此，递归过程不会导致循环定义

以(factorial n)为例

- 当参数n足够小时(=1时)，factorial直接有计算结果
- 否则，给出等式，并且保证等式右边的参数值更小

$$factorial(n) = \begin{cases} 1 & if\ n = 1 \\ n * (factorial(n-1)) & otherwise \end{cases}$$

把等式直接翻译为Scheme代码

```
#lang sicp

(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (factorial 4)
24
>
```

递归的思想

- 使用“规模”表示当前参数和递归出口之间的距离
- 当问题足够简单时，直接给出结果
- 当问题较为复杂时，假定任何规模较小的同一问题已解决，使用这些已解决的规模更小的同一问题以及已有的过程，来解决当前问题
- 递归算法只需给出“终点”和通用的“中间步骤”，无需给出“每一步”
- 递归等式描述了“中间步骤”，递归出口描述了“终点”
- 优点：清晰，简单，易于构造

保存中间信息

- 递归等式中不会涉及计算中每一步的那些信息
- 但是，递归过程在执行时会不断的“深入”
- 根据代换模型，这些信息会存在表达式中，导致整个表达式越来越长
- 这些信息就是递归过程执行时需保存的中间信息，由编译器/解释器负责保存。程序员无需关注

例子

- (factorial 4)
- (* 4 (factorial 3))
- (* 4 (* 3 (factorial 2)))
- (* 4 (* 3 (* 2 (factorial 1))))
- 这里的*4, *3和*2就是需要保存的中间信息

递归算法的实现

- 核心是写出递归等式和递归出口
- 递归等式和递归出口可以直接转化为Scheme代码
- 递归算法实现为条件语句，递归出口实现为一个条件判断以及分支，递归等式实现为另外的若干个分支

课堂思考题：

- 实现过程(f a b)，假定a和b已经是正整数且$a < b$。则(f a b)计算a+(a+1)+...+b的值
- 实现过程(g a b)，假定a和b已经是正整数且$a < b$。则(g a b)计算a*(a+1)*...*b的值

斐波那契数列

- 0，1，1，2，3，5，8，...
- 数列的第0项和第1项分别是0和1
- 从第2项起，每一项是之前两项的和
- 注意，数列的首元素被称为第0项，而不是第1项

根据定义，构造递归等式计算数列第n项，并几乎原样翻译为Scheme

$$fib(n) = \begin{cases} n & if\ n = 0\ or\ n = 1 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

```
#lang sicp

;斐波那契数列
(define (fib n)
  (cond ( (= n 0) 0)
        ( (= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (fib 6)
8
>
```

大纲

引入

(factorial 4)的求值过程

- (factorial 4)
- (* 4 (factorial 3))
- (* 4 (* 3 (factorial 2)))
- (* 4 (* 3 (* 2 (factorial 1))))
- 这里的4、3和2，以及3次乘法，都需要在程序执行的过程中保存起来。在计算(factorial 1)之后，再由一种机制自动做这3次乘法
- 对一些问题，可以优化递归算法，不再产生这些中间信息

迭代的设计思想

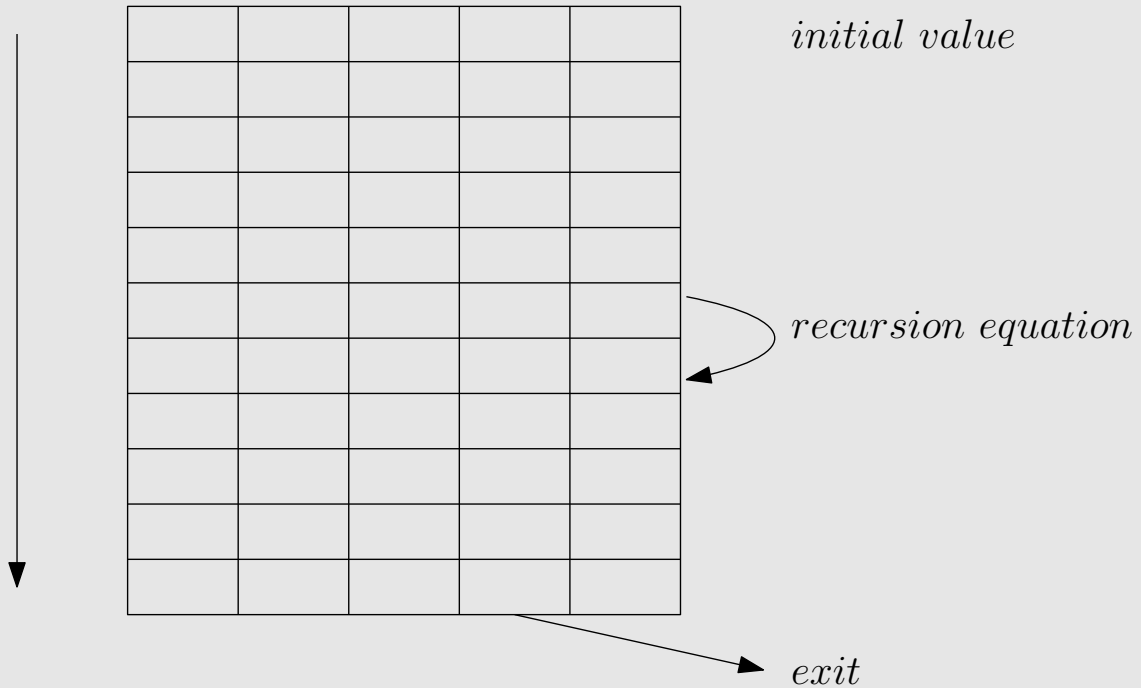
- 优点：完成和递归过程同样的任务，但不会产生中间信息
- 缺点：需要人工设计过程，而不能把递归等式直接翻译为迭代程序

设计思想

- 在使用迭代法解决问题时，事实上是不断地在填一张二维表
- 表的每一行的列数相同。每一列都有自己的含义。表格的每一行的各个列之间都满足一定的关系
- 调用过程，就是填写表的第一行
- 过程的执行，就是不断地基于表格的当前行，填写表格下一行
- 当表格当前行的某些列满足特定条件时，执行终止，此时当前行的某一列就是计算结果，

迭代

- 迭代算法事实上还是实现为递归算法
- 递归等式规定了如何从上一行计算出下一行
- 递归出口规定了计算何时结束以及计算结果
- 迭代程序的执行过程中，使用到的信息只有固定数目的参数，而不会产生额外信息



使用迭代计算阶乘

如何通过二维表格法计算阶乘，以计算6!为例

- 表格的一行有三列result、counter和counterBound
- 这三列满足性质：$result = counter!$，且counterBound总为6
- 初始从$1! = 1$开始，第一行result=1，counter=1和counterBound=6
- 假定当前行各列为result=$a!$、counter=a，counterBound=6
- 下一行是用来计算$(a+1)!$的，因此下一行的result的值等于本行的result乘以(counter+1)，下一行的counter等于本行的counter加1，下一行的counterBound不变。即，下一行result=$a! * (a+1)$、counter=$a+1$，counterBound=6
- 我们的计算保证了每一行中$result = counter!$
- 当计算来到$counter = 6$这一行时，此时$result = counter! = 6!$即为计算结果。此时结束递归，阶乘的计算结果即为此时result的值
- 我们遇到的第一个当某个参数足够大时结束执行的递归算法例子

二维表格如下

result=counter!	counter	counterBound
1	1	6
1*(1+1)=2	2	6
2*(2+1)=6	3	6
6*(3+1)=24	4	6
24*(4+1)=120	5	6
120*(5+1)=720	6	6

(factorial 6)的计算结果为720

- 可以看出，迭代算法的设计往往需要程序员额外引入变量和计算过程。无法直接把递归等式翻译为迭代算法

根据处理二维表格的方法，构造算法如下

- 最开始的函数调用用来填写二维表的第一行：(factorial-iter 1 1 n)
- 递归等式用来从上一行构造下一行：(factorial-iter result counter counterBound) ⇒ (factorial-iter result*(counter+1) counter+1 counterBound)
- 递归出口用来结束执行：当counter=counterBound时离开表格，此时计算结果为result

```
#lang sicp

;迭代计算阶乘
;满足result=counter!，且最终要计算counterBound的阶乘
(define (factorial-iter result counter counterBound)
  (if (= counter counterBound)
      result
      (factorial-iter (* (+ counter 1) result)
                      (+ counter 1)
                      counterBound)))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (factorial-iter 1 1 4)
24
>
```

递归计算过程和迭代计算过程

- 如果求值(f n)时，求值的每一步都可以用固定数目的状态变量描述，则这个求值的过程称为迭代计算过程（iterative process）
- 如果求值(f n)时，求值的步骤会产生需要额外保存的数字和运算，则这个求值的过程称为递归计算过程（recursive process）

注意

- 1.1节中定义的过程英文名为procedure。1.1节中定义的过程和迭代/递归计算过程仅仅是中文翻译时都使用了“过程”这两个字，其实是两个东西
- 一个过程（procedure）是递归的，如果其使用自己定义自己。仅需要看代码，无需看代码如何执行
- 一个计算过程是递归的（recursive process），如果求值过程中不断产生额外的信息。这需要看过程的执行

为了叙述方便，将有着迭代计算过程（iterative process）的过程（procedure）称为迭代过程

注释

- 当一行文字以;开头时，Scheme编译器不会编译这行文字
- 这样的文字称为注释
- 注释的作用是解释代码
- 在过程的开头要写注释，说明过程是做什么的，其参数各自代表什么含义
- 如果过程较为复杂，还需要在注释中写出程序的算法
- 如果过程体较长，还需要写注释说明每部分的任务
- 写注释方便自己和其他人。作者自己也是会忘掉细节的

包装迭代算法

- 迭代算法往往引入额外变量。用户不需要知道这些额外变量
- 用户只需要计算$n!$，到底以什么样的具体方式计算$n!$，用户是不关心的，这一点也不应该暴露给用户
- 因此，我们构造(factorial n)，在其中调用(factorial-iter 1 1 n)计算$n!$
- 提供给用户使用的过程是(factorial n)

```
#lang sicp

;迭代计算阶乘
;满足result=counter!，且最终要计算counterBound的阶乘
(define (factorial-iter result counter counterBound)
  (if (= counter counterBound)
      result
      (factorial-iter (* (+ counter 1) result)
                      (+ counter 1)
                      counterBound)))
(define (factorial n)
  (factorial-iter 1 1 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (factorial 4)
24
>
```

大纲

引入

(fib 43)的运行的时间就已经超过了1分钟

- (fib 43)=433494437
- 计算时间的急剧增加不光来自于计算结果数字较大
- 事实上，可以实现另一种递归算法，一瞬间就可以计算出(fib 43)，甚至(fib 100)

为此，我们需要知道

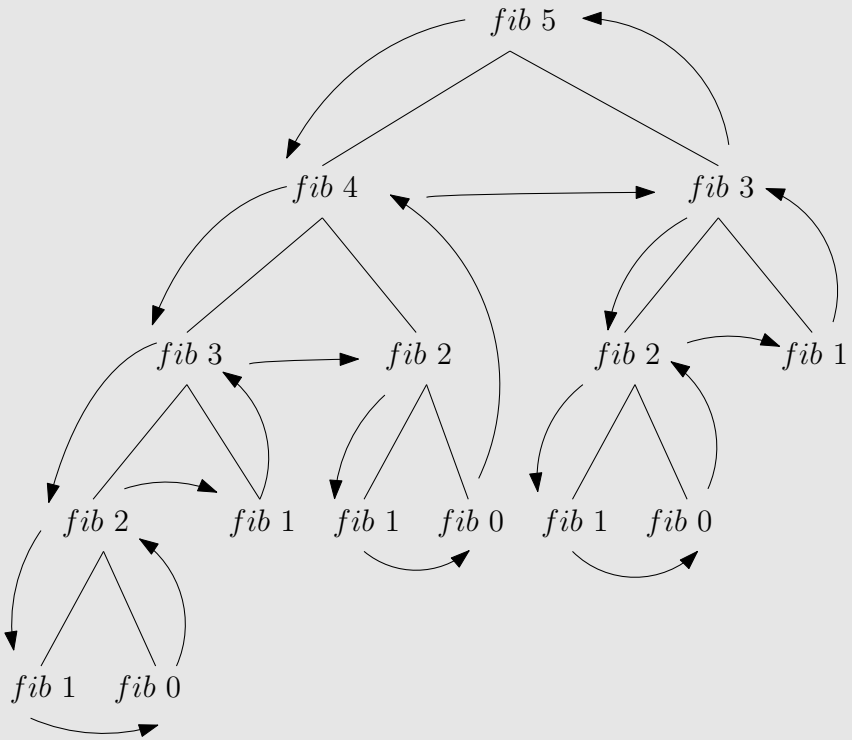
- 为什么(fib n)的计算时间会随着n的增加急速增加
- 什么样的递归算法可以让这样的增加不发生

(fib 5)的求值历程：多个子过程可求值时，假定优先求值最左的子过程

- (fib 5)，标红色的为当前要被求值的过程
- (+ (fib 4) (fib 3))
- (+ (+ (fib 3) (fib 2)) (fib 3))
- (+ (+ (+ (fib 2) (fib 1)) (fib 2)) (fib 3))
- (+ (+ (+ (+ (fib 1) (fib 0)) (fib 1)) (fib 2)) (fib 3))
- (+ (+ (+ (+ 1 (fib 0)) (fib 1)) (fib 2)) (fib 3))
- (+ (+ (+ (+ 1 0) (fib 1)) (fib 2)) (fib 3))
- (+ (+ (+ 1 (fib 1)) (fib 2)) (fib 3))
- (+ (+ (+ 1 1) (fib 2)) (fib 3))
- (+ (+ 2 (fib 2)) (fib 3))
- (+ (+ 2 (+ (fib 1) (fib 0))) (fib 3))
- (+ (+ 2 (+ 1 (fib 0))) (fib 3))
- (+ (+ 2 (+ 1 0)) (fib 3))
- (+ 3 (fib 3))
- . . .

(fib 5)求值过程的图形化表示

- 计算(fib 5)要先计算出(fib 4)和(fib 3)，所以在(fib 5)下面画(fib 4)和(fib 3)。(fib 5)到(fib 4)的箭头表示要求值(fib 5)需要先求值(fib 4)。(fib 4)到(fib 3)的箭头代表求值(fib 4)之后需要去求值(fib 3)。(fib 3)到(fib 5)的箭头代表求值完(fib 3)之后就可以根据(fib 4)和(fib 3)求值(fib 5)了
- 图上每个点都被求值了一次



根据上一页的图，统计计算(fib n)时，从(fib n)这个点及向下的图中的(fib 0)和(fib 1)节点个数

- 求值(fib 0)和(fib 1)：1个
- 求值(fib 2)：(fib 0)+(fib 1)，1+1=2=(fib 3)个
- 求值(fib 3)：(fib 1)+(fib 2)，1+2=3=(fib 4)个
- 求值(fib 4)：(fib 2)+(fib 3)，前者(fib 3)个，后者(fib 4)个，共(fib 3)+(fib 4)=(fib 5)个
- 不难证明，对更大的n，求值(fib n)：(fib n-1)+(fib n)=(fib n+1)个
- 如果可以给出(fib n+1)的数值，我们就可以估算计算(fib n)所需的计算次数的下界

- 教材练习1.13：fib(n)=$\frac{a^n-b^n}{\sqrt{5}}$，其中$a=\frac{1+\sqrt{5}}{2}, b=\frac{1-\sqrt{5}}{2}$
- $\sqrt{5}\approx 2.236$，因此$-1<b<0$
- 显然，$\frac{-1}{\sqrt{5}}<\frac{-b^n}{\sqrt{5}}<\frac{1}{\sqrt{5}}$，且n足够大时趋向于0
- $\frac{a^n}{\sqrt{5}}+\frac{-1}{\sqrt{5}}<fib(n)=\frac{a^n}{\sqrt{5}}+\frac{-b^n}{\sqrt{5}}<\frac{a^n}{\sqrt{5}}+\frac{1}{\sqrt{5}}$
- 因此，$fib(n)$的值约等于$\frac{a^n}{\sqrt{5}}$

算法改进思想

- 计算(fib n)的过程中要应用接近$\frac{a^n}{\sqrt{5}}$个过程
- 然而不同的过程应用仅有(fib 0), (fib 1), ... , (fib n)，共n+1个
- 显然，大量的过程被重复计算了
- 这种重复计算来自递归定义

新算法的设计思想

- 不再重复计算已经计算过的内容

如何做到这一点

一般性的设计思路

- 将计算过的子过程的值保存在某个地方
- 当需要递归计算时，先去查询之前是否计算过这样的值
- 如果计算过，就直接返回存储好的值。否则，才会实际运行子过程
- 这个方法的问题是，我们目前没有学习如何“存储”
- 可能被递归调用的子过程数目是任意的，因此需要一种能存储和查询任意多信息的机制
- 伏笔：在第三章，我们会学习“存储”信息，之后就可以实现这个机制。这种算法被称为动态规划算法

针对本问题的设计思路

- 具体问题具体分析
- 幸运的是，斐波那契数列问题可以被实现为一个迭代算法，只使用固定的信息就能够计算
- 新的算法速度更快，其加速来自于无需重复计算

“从下向上”的算法

在之前的递归计算法中

- 计算(fib 5)需计算(fib 4)和(fib 3)，计算(fib 4)需计算(fib 3)和(fib 2)
- 在图片上，计算的顺序是从上到下，这导致很多过程被反复计算

“从下到上”的计算顺序

- 先用(fib 0)和(fib 1)计算出(fib 2)
- 然后，可以安然的忘掉(fib 0)，因为计算(fib 3), (fib 4)...时无需(fib 0)
- 使用(fib 1)和(fib 2)计算出(fib 3)
- 然后，忘掉(fib 1)
- 使用(fib 2)和(fib 3)计算出(fib 4)
- ...
- 显然，每次只需要记住两个数，算出第三个数

使用从下向上的算法计算(fib 5)的计算流程如图

$$\begin{array}{ccccccc} 0 & + & 0 & = & 1 & & \\ (fib\ 0) & & (fib\ 1) & & (fib\ 2) & & \\ & & 0 & + & 1 & = & 2 \\ & & (fib\ 1) & & (fib\ 2) & & (fib\ 3) \\ & & & & 1 & + & 2 \\ & & & & (fib\ 2) & & (fib\ 3) \\ & & & & & & 2 \\ & & & & & & (fib\ 3) \end{array} \quad \begin{array}{ccc} = & 3 & \\ & (fib\ 4) & \\ & & \\ & & \\ = & 5 & \\ & (fib\ 4) & \\ & & \\ & & \\ + & 3 & = \quad 5 \\ & (fib\ 4) & (fib\ 5) \end{array}$$

- 每一行固定3列
- 计算(fib n)时，一共有n-1行
- 因此，可以实现为迭代算法

使用迭代计算斐波那契数列

通过二维表计算(fib 5)

- 表格的一行有四列idx、curValue、nextValue和idxBound
- curValue=(fib idx)是斐波那契数列的第idx项，nextValue=(fib idx+1)是斐波那契数列的第idx+1项。idxBound=5
- 注意，斐波那契数列的首元素称为第0项
- 表格的第一行：idx=0，curValue=(fib idx)=0，nextValue=(fib idx+1)=1，idxBound=5
- 从当前行构造下一行时，下一行的idx为当前行idx+1，下一行的curValue为当前行nextValue，下一行的nextValue为当前行curValue加当前行nextValue
- idx=idxBound时结束计算，此时curValue为计算结果

使用迭代法计算(fib 5)的二维表格如下

idx	curValue=fib(idx)	nextValue=fib(idx+1)	idxBound
0	0	1	5
1	1	0+1=1	5
2	1	1+1=2	5
3	2	1+2=3	5
4	3	2+3=5	5
5	5	3+5=8	5

(fib 5)的计算结果为5

根据处理二维表格的方法，构造算法如下

```
#lang sicp

;迭代计算斐波那契数列
;arguments: idx, fib(idx), fib(idx+1)
(define (fib-iter idx curValue nextValue idxBound)
  (if (= idx idxBound)
      curValue
      (fib-iter (+ idx 1)
                nextValue
                (+ curValue nextValue)
                idxBound)))

(define (fib n)
  (fib-iter 0 0 1 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (fib 5)
5
>
```

- 并不是每一个包含重复计算的递归过程，都可以像斐波那契数列一样写成迭代算法

大纲

引入

- 当过程较复杂时，有时会定义辅助过程，如square
- Scheme不允许多个过程有相同的名字
- 可能你需要编写多个过程，其中一些过程会使用一些相近但不同的辅助过程
- 如何让这些辅助过程的名字不会彼此影响
- 不好的解决方法：为辅助过程取唯一的名字

本小节介绍的解决方法

- 把辅助过程的定义搬到使用他的过程之内
- 根据Scheme的语法，其他函数看不到这个辅助过程，因此无需顾虑重名

牛顿法的引入

- 计算机支持的数字运算只有加减乘除
- 计算机有时需要完成复杂的计算任务，例如计算π、$\sqrt{2}$、$\sqrt[3]{5}$、$sin(x)$等
- 一些计算任务可以转化为求解方程
- 因此，如果我们可以编写程序让计算机可以解出方程的根，我们就可以完成这些任务
- 例子：计算$\sqrt{2}$这个任务，就可以转化为计算方程$x^2 - 2 = 0$的根

注意

- 我们需要的并不是像我们中学课堂学的那样计算方程的根，毕竟方程$x^2 - 2 = 0$的根是$\sqrt{2}$
- 我们需要计算机算出方程$x^2 - 2 = 0$的根的确切数值

这里要注意计算机和我们以前了解的数学的一个区别

- 数学上我们可以使用定义来描述一个数
- $\sqrt{2}$就是大于0且平方为2的那个数
- $\sqrt{2}$就是两条直角边长度都是1的直角三角形的斜边长度
- 然而，计算机无法理解这种描述，它只能执行代码

然而，从加减乘除出发，如何计算$\sqrt{2}$呢

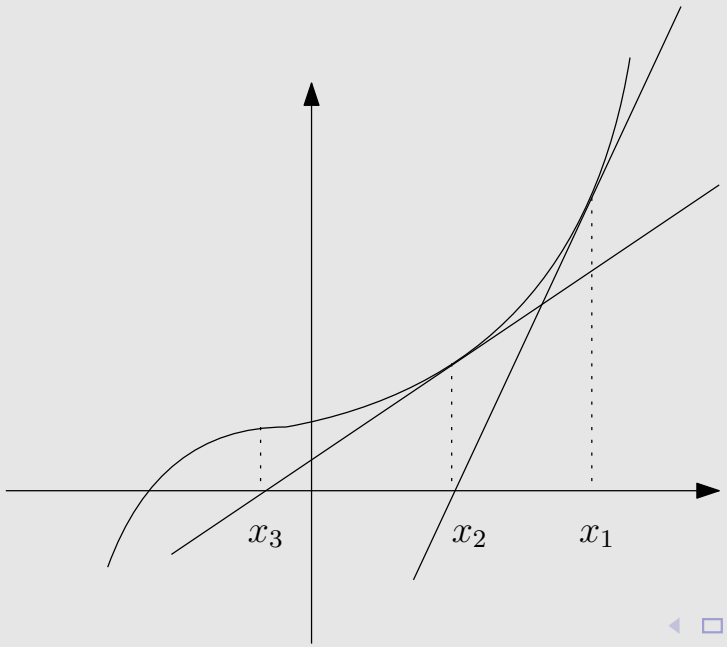
牛顿法

- 牛顿法是一种求解方程的根的方法
- 对相当一部分曲线，可以近似地计算出根
- 思路：不断逼近方程的根
- 注意，计算机在计算实数时，做不到百分之百精确，只能做到逼近
- 可以尝试在DrRacket里计算(+ 0.1 0.2)，结果是否是0.3呢
- 伏笔：计算机如何存储实数
- 伏笔：在理论上，计算机不可能做到准确地存储所有实数

牛顿法原理的图示

如何求解曲线$f(x)$对应的方程

- 猜测值x_1，从x_1计算x_2，从x_2算x_3,...直到算到足够接近方程解的值
- x_2：$f(x_1)$的切线和x轴的交点，可以看到x_1处竖线，x轴和$f(x_1)$的切线构成直角三角形
- x_3：$f(x_2)$的切线和x轴的交点，...
- x_1，x_2，...越来越靠近f(x)=0的根
- 当某个f(x_k)足够接近0时，认为x_k就是方程的根



- 牛顿法在计算的过程中无需了解f(x)的特征，例如是几次方程，是否有加减乘除以外的操作(如sin)
- 牛顿法未必适合求解每一个方程的根，但对很多方程它是适用的

牛顿法如何从x_i计算x_{i+1}

- 满足等式$f(x_i) + f'(x_i)(x_{i+1} - x_i) = 0$
- 因此，$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- 这里$f'(x)$为$f(x)$的导数，或称斜率

使用牛顿法计算方程的根

构造迭代算法求解方程f(x)=0的根

- 二维表格的第一行存放初始猜测的值
- 二维表格的每一行存储当前猜测的值x
- 每一轮先检测当前猜测值x是否足够好（即f(x)的值和0的误差是否足够小）
- 如果是，则当前猜测值x是计算结果
- 否则，开始下一轮(填写下一行)，新的猜测值为$x - \frac{f(x)}{f'(x)}$

当前我们只能做到针对每一个f(x)设计一个算法，而无法构造一个通用的过程，输入方程f，自动求解

- 伏笔：在1.3节我们将实现这个过程

使用牛顿法计算平方根

构造过程(sqrt n)计算n的平方根

- 待求解的方程为$f(x) = x^2 - n$
- 因此，导数$f'(x) = 2x$
- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - n}{2x_i} = \frac{x_i + \frac{n}{x_i}}{2}$
- 不妨令初始猜测值$x_1 = 1$

代码实现

```
#lang sicp

;求平方根
(define (sqrt-iter x n)
  (if (good-enough? x n)
      x
      (sqrt-iter (improve x n) n)))

(define (improve x n)
  (average x (/ n x)))

(define (average n y)
  (/ (+ n y) 2))

(define (good-enough? x n)
  (< (abs (- (square x) n)) 0.0001))

(define (square n)
  (* n n))

(define (sqrt n)
  (sqrt-iter 1.0 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (sqrt 2)
1.4142156862745097
>
```

- (sqrt-iter x n)使用迭代计算$\sqrt{n}$，当前猜测值为x
- abs是Scheme默认提供的过程，计算绝对值
- improve、average、good-enough?和square都是辅助计算的过程

过程抽象

- sqrt把improve等辅助过程视为实现好的黑箱子，直接使用
- 未来可以把improve等辅助过程替换为更好的实现，而这一切对sqrt过程而言是透明的，或者说，无法察觉的

sqrt过程也应该直接提供给用户，用户无需了解任何细节即可直接使用

- 然而，improve等过程可以被别的过程访问，这导致sqrt的内部实现被泄露出来
- 可以通过内部定义技术解决这个问题

课堂思考题：使用牛顿法求方程$x^3 - a = 0$的解

内部定义

- 过程体中可以定义另一个过程
- 新的过程定义格式：(define (过程名 参数) (过程定义1) . . . (过程定义k) expr)
- (过程定义1) . . . (过程定义k)是k个局部于当前定义过程的内部过程
- 当前定义过程的过程体是expr
- expr可以使用这些内部定义的过程

将good-enough?等过程的定义移入sqrt过程的定义中

```
#lang sicp

;求平方根
(define (sqrt n)
  (define (sqrt-iter x n)
    (if (good-enough? x n)
        x
        (sqrt-iter (improve x n) n)))
  (define (improve x n)
    (average x (/ n x)))
  (define (average n y)
    (/ (+ n y) 2))
  (define (good-enough? x n)
    (< (abs (- (square x) n)) 0.0001))
  (define (square n)
    (* n n))
  (sqrt-iter 1.0 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (sqrt 2)
1.4142156862745097
>
```

形式参数的作用域

- 过程sqrt和过程average都有参数n
- 这两个n是不一样的
- 前者是要开方的参数，而后者是要求平均值的两个参数之一
- 为社么Scheme不会混淆这两者

形式参数的作用域

- 过程的参数x在过程体中有效，也在这个过程内部定义的其他过程中有效
- 如果外层过程和内部过程都定义了参数x，则在内部过程中生效的x是内部函数自己定义的那个
- 尽管如此，为了清晰起见，内部过程的参数最好不要和外层过程的参数名字相同

大纲

- 递归
- 递归的改进一：迭代
- 递归的改进二：去掉重复计算
- 内部定义
- 增长阶
- 递归的改进三：二分
- 结束

引入

- 我们知道了递归的斐波那契数列算法运行慢，迭代的斐波那契数列算法运行快
- 递归的斐波那契算法计算(fib n)需要执行至少$(\frac{1+\sqrt{5}}{2})^{n+1}/\sqrt{5}$次运算
- 迭代的斐波那契算法计算(fib n)需要依次计算出(fib 2)、(fib 3)、...，(fib n)。每次计算需要更新一行的三个参数。共执行(n-1)*3次运算
- 程序执行会消耗资源：时间资源，空间资源
- 递归的斐波那契算法相比迭代的斐波那契算法坏，因为它执行更慢
- 什么是执行慢：完成同样的任务，需要的运算次数更多
- 当n足够大时，$(\frac{1+\sqrt{5}}{2})^{n+1}/\sqrt{5}$远远大于(n-1)*3

本节正式定义算法的好坏：增长阶（消耗的资源量所处数量级）的高低

- 时间增长阶：程序执行消耗的时间资源
- 空间增长阶：程序执行消耗的空间资源

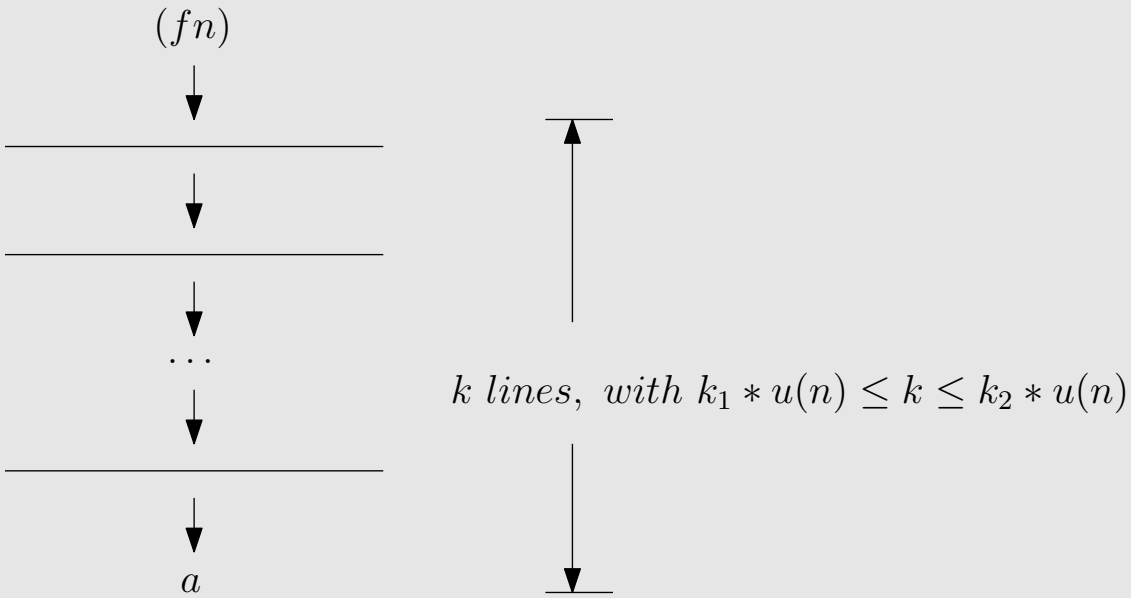
时间增长阶

- 用R(n)定义在输入的规模为n的情况下，程序执行占用的资源数量。对时间，资源指指令的数目，可以简单理解为代换模型执行的行数
- 输入规模：依问题而定，往往是参数的大小，数据的长度等
- 注意，时间增长阶中的时间指的是执行的指令数目(本章中是代换模型执行的次数)，而不是具体的时间

时间增长阶

- 我们称过程f有着$\Theta(u(n))$的时间增长阶，如果存在与n无关的整数k_1和k_2，使得对每个参数n，(f n)在代换模型下执行的行数在$k_1 * u(n)$和$k_2 * u(n)$之间
- 时间增长阶更在意数量级，即是$\Theta(n)$还是$\Theta(n^2)$
- 由于$\Theta(n) = \Theta(2 * n)$，所以时间增长阶不在意时间是否变化了固定的倍数

时间增长阶的图示，以时间增长阶为$\Theta(u(n))$为例



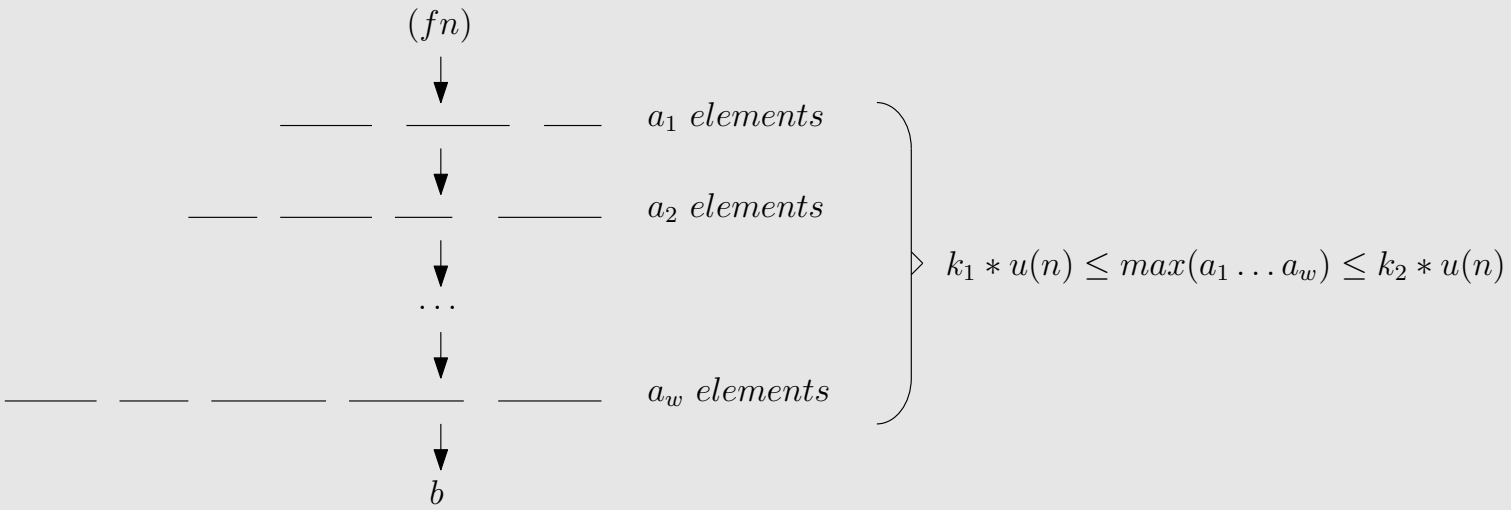
空间增长阶

- 空间增长阶关注程序占用的存储空间
- 在当前章节可以理解为代换模型执行过程中最长的一行的函数应用+参数的数目

空间增长阶

- 我们称过程f有着$\Theta(u(n))$的空间增长阶，如果存在与n无关的整数k_1和k_2，使得对每个参数n，(f n)在代换模型下每行的存储空间的最大值在$k_1 * u(n)$和$k_2 * u(n)$之间
- 类似时间增长阶，空间增长阶关注的也是数量级

空间增长阶的图示，以空间增长阶为$\Theta(u(n))$为例



- 在本章中，每个元素都是数字，因此本章的迭代算法的空间增长阶是常数，因为参数数目固定
- 下一章我们将引入表，一个参数可以是一个包含多个数据的表，因此下一章的迭代算法未必空间增长阶为常数
- 相对时间，空间往往重要性稍逊

- 同一个算法对不同输入的执行“时间”可能是不同的
- 例如，如果算法的任务需要是在一群人里找人，实现方法是把人排为一列点名。如果要找的人排在第一个，那一步就执行完成；如果要找的人不在队列里，则需要把队列都点名完毕
- 第一种情况的执行时间是$\Theta(1)$，第二种情况的执行时间是$\Theta(n)$
- (时间/空间)增长阶这个概念往往也有最坏情况下的增长阶和平均情况下的增长阶等细致区分
- 一般提到增长阶时，往往指的是最坏情况下的增长阶

例子：factorial过程的时间和空间增长阶

考察factorial过程的执行，以递归的(factorial 3)为例

- (factorial 3)
- (if (= 3 1) 1 (* 3 (factorial (- 3 1))))
- (* 3 (factorial 2))
- (* 3 (if (= 2 1) 1 (* 2 (factorial (- 2 1)))))
- (* 3 (* 2 (factorial 1)))
- (* 3 (* 2 (if (= 1 1) 1 (* 1 (factorial (- 1 1))))))
- (* 3 (* 2 1))
- (* 3 2)
- 6

时间增长阶：计数(factorial n)的执行行数

- 写下(factorial n)
- 从(factorial k)到(factorial k-1)需2行。这样的减小共n-1次，直到(factorial 1)
- 把(factorial 1)代换为1需要2行
- 之后做n-1次乘法，每次乘法代换1行
- 求值(factorial n)共需2*(n-1)+2+(n-1)=3n-1行

因此，时间复杂度是线性的，即$\Theta(n)$

- 课堂思考题：迭代的阶乘算法的时间增长阶是多少

空间增长阶：计数(factorial n)执行时元素最多的行的元素数目

- 参数每减小1，就多一次乘法和一个被乘数，即多2个元素
- 元素最多的一行是(* n (* n-1 ... (* 2 (if (= 1 1) 1 (* 1 (factorial - 1 1))))，共2*(n-1)+11=2n+9个元素(这里把if、过程和数字都算一个元素，括号没算)

因此，这个算法的空间增长阶都是线性的，即$\Theta(n)$

大纲

引入

递归的前两个改进还不够

- 从递归和迭代的阶乘算法可以看出，迭代技术并没有本质上让算法运行的更快
- 去掉重复计算这个改进，仅仅只是去掉了冗余的计算过程，但对非冗余部分，并没有改进

本小节我们引入递归的另一种改进

- 是一种让递归等式的难度从本质上降低的方法
- 而且，很好理解

二分法

要求

- 递归等式的右边只有一个递归调用
- 每次递归调用，参数的规模成比例下降，一般是下降一半

求幂

计算a^b的递归等式如下

$$a^b = \begin{cases} 1 & if\ b = 0 \\ a * a^{b\text{-}1} & otherwise \end{cases}$$

根据递归等式，立刻构造递归算法如下

```
#lang sicp

;求幂
(define (expt base n)
  (if (= n 0)
      1
      (* base (expt base (- n 1)))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (expt 5 4)
625
>
```

使用迭代计算求幂

- 表格的一行有四列base、curN、product和nBound
- product=$base^{curN}$是已计算出的部分幂函数的结果。nBound是要计算的幂函数的指数
- 第一行填入curN=0，product=1
- 从当前行构造下一行时，下一行的base和nBound不变，下一行的curN为当前行的curN加1，下一行的product为当前行的base乘以当前行的product
- curN=nBound时结束计算，此时curValue为计算结果

以(expt 5 4)为例，二维表格如下

base	curN	$product = base^{curN}$	nBound
5	0	1	4
5	1	5*1=5	4
5	2	5*5=25	4
5	3	5*25=125	4
5	4	5*125=625	4

(expt 5 4)的计算结果为625

根据处理二维表格的方法，构造迭代算法如下

```
#lang sicp

;迭代求幂
(define (expt base n)
  (define (expt-iter base curN product nBound)
    (if (= curN nBound)
        product
        (expt-iter base
                   (+ curN 1)
                   (* base product)
                   nBound)))
  (expt-iter base 0 1 n))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (expt 5 4)
625
>
```

- 不难证明，递归和迭代的幂函数算法的时间增长阶都是线性的
- 在算法的思想不变的情况下，递归改为迭代不能加速算法的增长阶
- 能否加速算法的时间增长阶

二分的求幂算法

如何二分

- 如果已知$a^{\frac{n}{2}}$，如何求a^n？进行一步平方运算即可
- $a^n = (square\ a^{\frac{n}{2}})$，先递归求出$a^{\frac{n}{2}}$的值b，而后(square b)=b*b即为所得
- $a^8 = (square\ a^4) = (square\ (square\ (square\ a)))$

对应的递归等式如下。注意递归等式有三个分支

$$a^n = \begin{cases} 1 & if\ n = 0 \\ (square\ a^{\frac{n}{2}}) & if\ n\ is\ even \\ a * a^{n-1} & otherwise \end{cases}$$

根据递归等式，构造递归算法如下

```
#lang sicp

;二分法计算幂
(define (fast-expt b n)
  (define (square x)(* x x))
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (fast-expt 5 4)
625
> (fast-expt 5 3)
125
>
```

注意

- $a^n = (square\ a^{\frac{n}{2}})$，而不是$a^n = a^{\frac{n}{2}} * a^{\frac{n}{2}}$
- 课堂思考题：如果根据后者写出递归算法，其时间增长阶是多少

fast-expt的时间增长阶

- 不妨令$c = 2^k$
- (fast-expt a c)通过一次if语句，变为(square (fast-expt a $\frac{c}{2}$))
- (fast-expt a c)的运行时间
- = (square (fast-expt a $\frac{c}{2}$))的运行时间+2 = ...
- = (square ... (square (fast-expt a 0))的运行时间+2*$log_2(c)$
- = (square ... (square 1)的运行时间+2*$log_2(c)$+1，共$log_2(c)$个square
- 每个square需2行。所以，fast-expt的运行时间为4*$log_2(c)$+1，时间增长阶为$\Theta(log_2(n))$，为对数

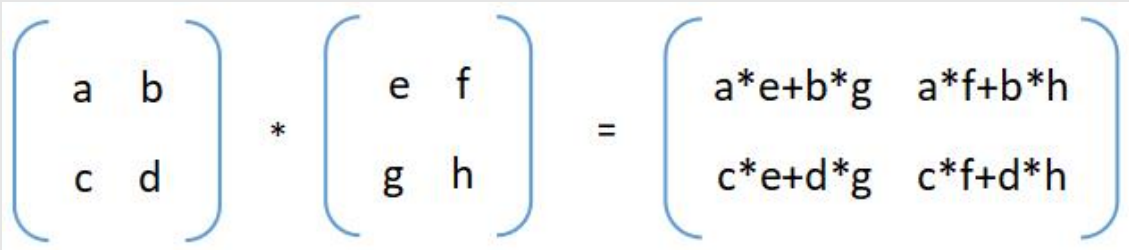
二分法确实减小了时间增长阶：线性时间 $\Rightarrow$ 对数时间

矩阵乘法与斐波那契数列

矩阵：一个矩阵包含若干行若干列，每个元素是一个数字

- 二阶矩阵：2行2列的矩阵，包含4个元素

二阶矩阵的乘法如下



不难证明，对$n \geq 1$，斐波那契数列满足如下矩阵等式

$$\begin{bmatrix} fib(n+1) & fib(n) \\ fib(n) & fib(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

课堂思考题：证明这一点

- 由于转化为幂，不难看出这是一个对数时间的斐波那契数列算法
- 当前我们难以简洁地实现这个算法，因为我们无法把一个2*2的矩阵表示为一个运算对象
- 伏笔：在第二章，我们将实现这个算法
- 据此，我们可以相信，斐波那契数列可以在对数时间内计算

最大公约数

最大公约数：两个数的公因子中最大的那个

- 令(gcd a b)为计算a和b的最大公约数
- 朴素的算法：从min(a,b)递减到2，遇到的第一个公约数就是a和b的最大公约数
- 算法的时间增长阶：线性
- 例如，如果a和b互质，则计算会从min(a,b)开始扫描，一直扫描到2

欧几里得算法

定义过程(gcd a b)计算a和b的最大公约数

- 假定$a > b$。令a=c*b+d
- 不妨令x是a和b的最大公约数。如果b和d存在大于x的公因数y，则y也是a的因数，不成立。因此，b和d的公因数小于等于x
- 令a=m*x，b=n*x，则m*x=c*n*x+d。因此，d=(m-c*n)*x，b和d的最大公因数为x
- a和b的最大公约数=b和d的最大公约数
- 递归等式：(gcd a b)=(gcd b d)，参数变小了
- 递归出口：当a是b的倍数时，(gcd a b)=(gcd b 0)=b

例子：gcd(108,40)=gcd(40,28)=gcd(28,12)=gcd(12,4)=gcd(4,0)=4

递归算法

```
#lang sicp

;最大公约数
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (gcd 108 40)
4
>
```

- 欧几里得算法的时间增长阶是什么
- 欧几里得算法的参数每轮都在减小，减小的幅度幅度是多少

- 前面已经讲述，(fib n)几乎等于$\frac{1}{\sqrt{5}} * (\frac{(1+\sqrt{5})}{2})^n$，是指数增加的

*Lamé*定理

- 如果欧几里得算法需要用k步计算出一对整数的最大公约数，则这对数中较小的那个数必然大于或者等于第k个斐波那契数

证明

- 令连续三次计算为(gcd a b) $\Rightarrow$ (gcd b c) $\Rightarrow$ (gcd c d)
- 由算法可知，存在x，使得b = x*c+d $\geq$ c+d
- 算法的计算序列：(gcd a_k b_k) $\Rightarrow \ldots \Rightarrow$ (gcd a_2 b_2) $\Rightarrow$ (gcd a_1 b_1) $\Rightarrow$ (gcd a_0 b_0)
- 显然$b_0 = 0 = fib(0)$，$b_1 \geq 1 = fib(1)$，$b_2 \geq b_0 + b_1 \geq fib(2)$, ...。证毕

因此，欧几里得算法的参数是按照比例减小的

成比例下降

如果

- 递归等式的右边只有一个递归调用
- 递归调用的参数成比例下降
- 递归等式右边其他运算(例如计算新参数，以及中间信息和对应的计算)使用常数时间，不妨令在常数b步之内完成

则算法f的时间增长阶为对数

- 证明，令每次参数下降为原来的$\frac{1}{a}$
- (f n)的计算时间=b+(f $\frac{n}{a}$)的计算时间=b+b+(f $\frac{n}{a^2}$)的计算时间 = ...
- = b*(log n)+(f 0) = b*(log n) + c
- 这里假定(f 0)为递归出口，使用常数时间c计算

成比例下降这个技术能够把算法的时间增长阶优化到对数

- 二分的幂算法，以及欧几里得算法，都是成比例下降的算法的例子

大纲

课堂总结

仅用1.1节的语法，我们就可以完成复杂的任务

- 递归：让算法可以重复工作
- 递归=递归等式+递归出口
- 迭代：用二维表格法设计算法
- 递归等式包含大量重复，去掉重复可以提速
- 增长阶：评价算法的标准
- 二分：如果递归等式有着特定的形式，则可以通过二分技术加速算法的执行

Scheme语法

- 内部定义
- 教材练习1.22：newline过程和display过程
- (newline)：打印一个换行
- (display a)：打印a的值
- 未讲的1.2.6小节中的(random a)过程：随机地返回一个小于a的非负整数

本节内容与其他知识的联系

在后续章节中会进一步进展的

- 第三章学习存储后，我们会通过存储已经求值的计算结果来加速算法
- 1.3节中，我们将实现一般的牛顿法，可以为任何方程求根，而不是只能求平方根
- 第三章学习存储后，我们将实现对数时间增长阶的斐波那契数列算法

1.2节涉及和导引的其他课程内容较多

递归

- 递归是计算机编程以及计算机科学中一个非常重要的思想

在编程方面

- 递归是编写算法的基础技能
- 在算法设计与分析课程中，使用递归来解决问题是一个基本的算法设计方法(称为分治法)
- 二分法也是算法设计与分析课程的内容之一

在计算机科学方面

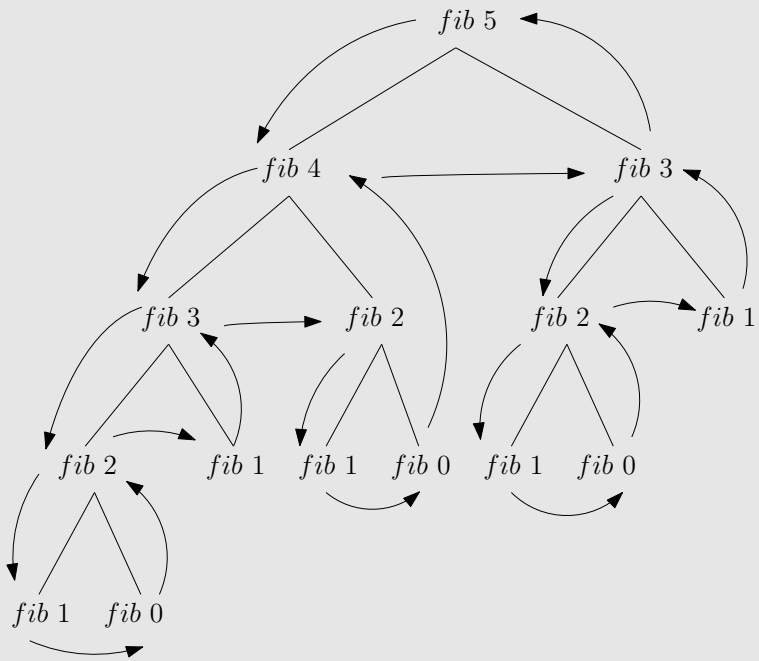
- 计算机科学一个基本的研究方向是，什么是计算的（理论）模型，如果这样的模型有很多，他们之间的能力强弱关系如何
- 例如，计算机最被广为接受样的理论模型称为图灵机
- “递归函数”也是计算机的理论模型，而且能力和图灵机“一样强”
- 这里的递归函数打了引号，因为和本章的递归函数有些不一样
- 本章的递归，在理论上称为原始递归
- 计算机科学中的递归函数=原始递归+极小运算符
- 专门有一门学科研究递归函数，叫做递归函数理论

以上的部分内容，可以在每周五上午2-4节的计算理论课程中学习到，欢迎感兴趣的同学来了解

- 教室：8教108
- 授课教师：也是我

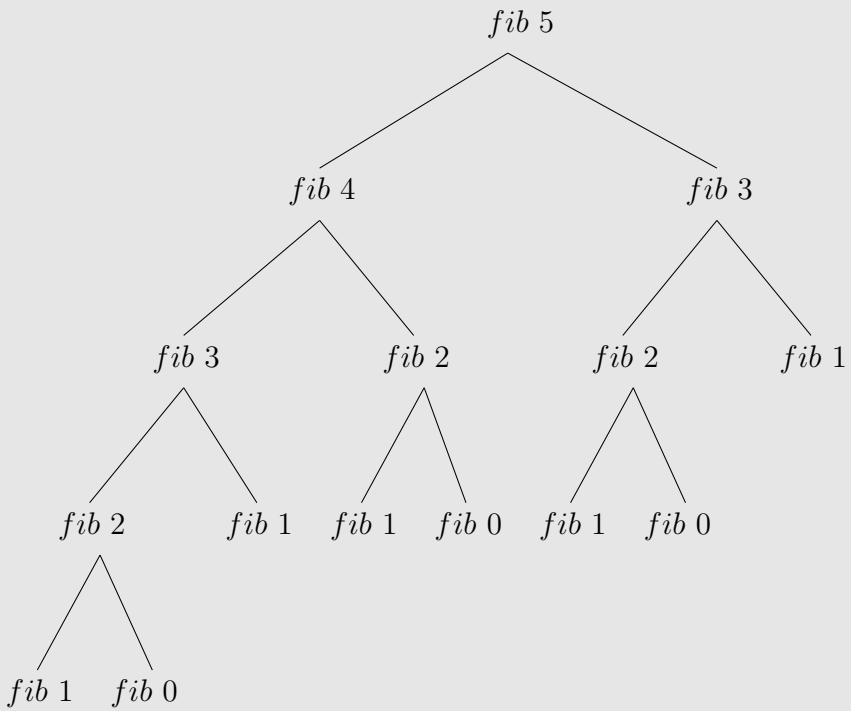
树

递归的斐波那契数列求解(fib 5)的流程如下



这里的这个图片(只算节点和直线边，不算箭头)称为树

树的例子


$fib\ 5$
$fib\ 4$
$fib\ 3$
$fib\ 3$
$fib\ 2$
$fib\ 2$
$fib\ 1$
$fib\ 2$
$fib\ 1$
$fib\ 1$
$fib\ 0$
$fib\ 1$
$fib\ 0$
$fib\ 1$
$fib\ 0$

计算机领域中的树

- 可以理解为将自然界的树倒过来
- 最高层节点称为根节点
- 每个节点都可以连接下一层的节点，每个低层节点不能被多于一个高层节点连接
- 被一条线段连接起来的高层节点称为父节点，低层节点称为子节点
- 根节点没有父节点
- 没有子节点的节点称为叶节点

在离散数学、数据结构等课程中，我们将学习树

计算机中数字的编码

本门课程通过Scheme为媒介，接触并了解计算机程序的设计思想

- Scheme为使用者提供了一个较为抽象的世界
- 但计算机实际运行在一个更为底层的世界上
- 在计算机组成原理，计算机体系结构，汇编语言等课程中，大家将了解计算机的硬件构成，处理器的工作原理，以及计算机底层提供给上层的编程语言
- 体现这两者区别的一点是数字的编码
- 在其他编程语言中，一个数据往往有着明确的大小以及明确的范围
- 例如，C语言中一个整数数据的表示范围是-2147483648到+2147483647
- 然而，在Scheme中，大部分情况下你无需关注这个界限。例如，试着执行(fast-expt 2147483647 10000)

整数的编码

- 计算机中的数据，在底层都是以二进制形式保存和处理的
- 一个二进制位称为1个bit（位），八个二进制位称为一个byte（字节）
- 计算机中的存储单位是字节。1K=1024字节，1M=1024K，1G=1024M
- 整数在计算机中使用一种称为补码的方式进行保存。在后续课程中会学习到
- 计算机中，每个整数或实数都需要被存储为特定长度的二进制串

实数的编码

- 一些实数是无限循环小数，例如$\frac{1}{3}$=0.33333...
- 一些实数甚至是无限不循环小数，例如π
- 显然，π这样的数字无法精确保存，而只能保存前若干位
- 计算机中使用一些特定的格式来存储实数，例如IEEE 754标准，在偏系统的课程会学习到

牛顿法

- 并不是为了学习牛顿法才特意只使用加减乘除来计算平方，而是计算机能支持的数字计算确实只有加减乘除(求余数可以认为是用除法实现的)
- 数值计算方法这门课，讲述了如何使用计算机来计算各种复杂的函数，以及求方程
- 牛顿法属于数值计算方法
- 数值计算方法(如牛顿法)和经典算法(如计算阶乘)有一个区别，前者在计算足够接近真正的解时停止，而后者会计算出真正的解

解方程

牛顿法可以求解大量方程近似的数值解。那这些方程准确的解如何求呢

- 对简单的方程，例如一元一次方程或者一元二次方程，我们可以使用求根公式
- 一些同学也许已经了解到了，对一元三次方程，也是有求根公式的
- 然而，对五次及以上的方程，没有求根公式
- 这一结论来自于伽罗瓦的结果
- 近世代数课程会涉及到这个内容

算法的增长阶和问题的增长阶

- 我们学习了使用增长阶来评估算法的时间和空间资源占用
- 通过斐波那契数列算法和求幂算法，我们看到了算法的时间增长阶可以被改进
- 和算法的时间增长阶相关的问题是，待求解的问题其固有的难度是什么
- 或者说，解决这个问题的所有算法，其时间增长阶的下界（最小的时间增长阶）是什么
- 乍一看固有难度是无法研究的，因为解决一个问题的算法有无穷多
- 问题本身固有的时间和空间增长阶是计算机科学的重要研究方向
- 计算复杂性这门课中，会学习这些知识

概率算法

PPT中未讲的1.2.6小节

- 很多问题固有的时间增长阶过高
- 概率算法是一种改进，可以较快地得到近似的结果
- 通过牺牲严格的正确性，换取算法不那么慢，进而可以实用
- 这样的故事你们会在自己的职业生涯中多次听到

预告

- 牛顿法是一套求解各种方程的方法，而不是只用来求解开平方
- 在1.3节，我们将引入高阶函数，实现一般的牛顿法
- 它可以已“一个方程”作为参数，灵活性大大增强

1.2节练习题

练习级

- 实现一个过程，输入数字n，计算1到n的和
- 教材练习1.6
- 教材练习1.8
- 教材练习1.10(Ackermann函数是一个非原始递归函数)
- 教材练习1.11
- 教材练习1.15：计算sin x

挑战级

- 教材练习1.12
- 教材练习1.16
- 教材练习1.25
- 教材练习1.26

- 系统的学习了程序编写
- 初次接触递归会不习惯，需多练习
- 有问题及时提问，以及和老师交流

下课