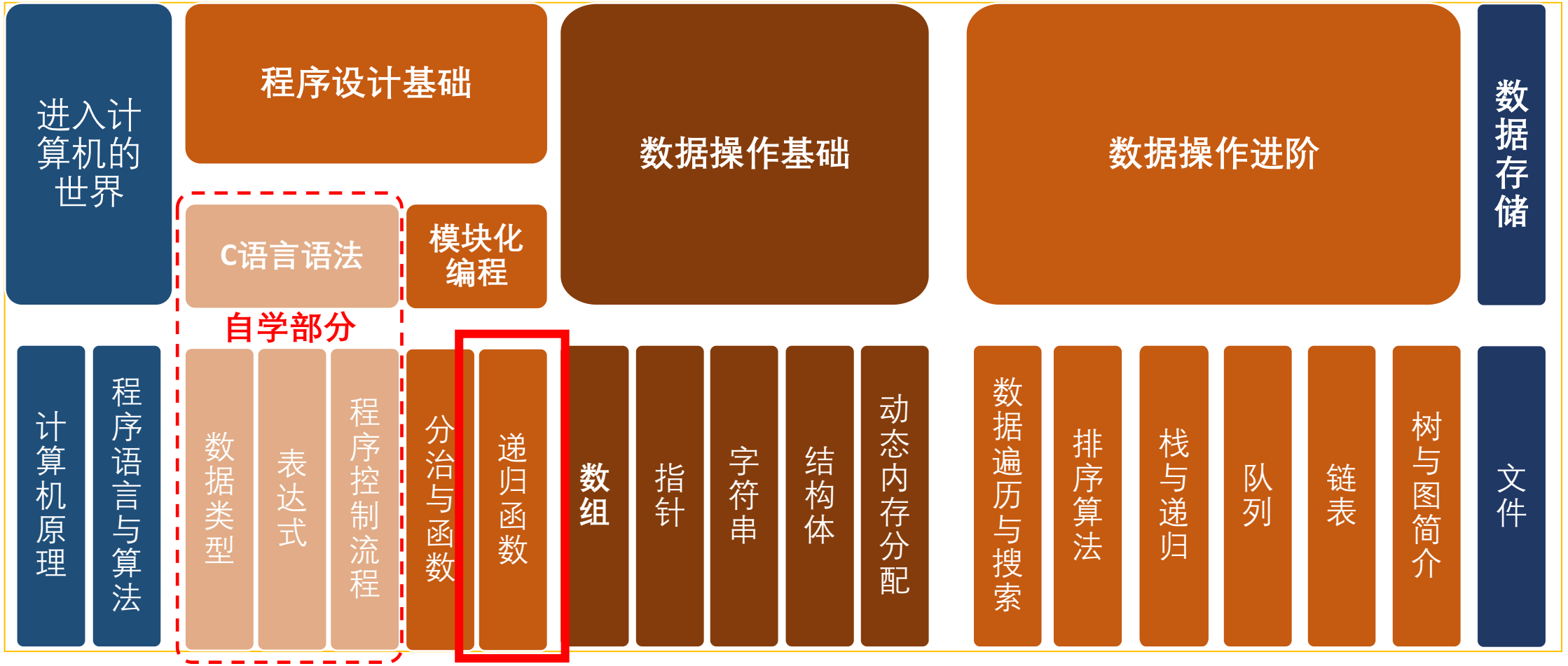
计算思维与实践

哈尔滨工业大学（深圳）
计算机科学与技术学院
大数据技术中心
张保权

课件.版权：哈尔滨工业大学.苏小红 sxh@hit.edu.cn
版权所有，违者必究

课程内容安排



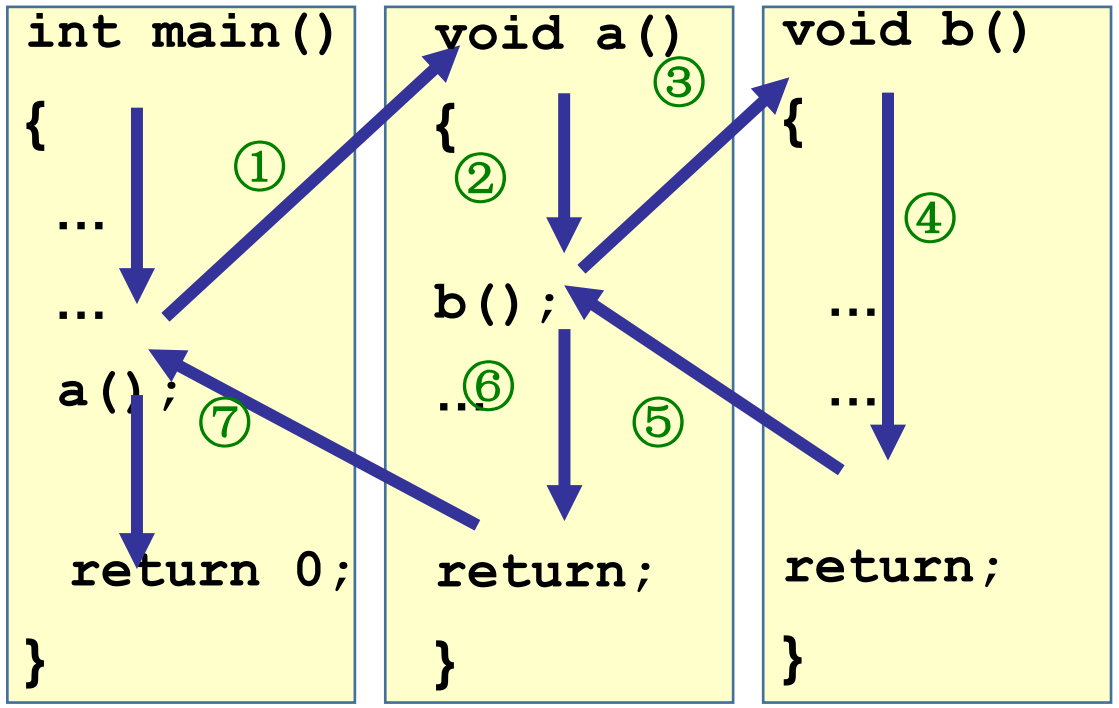
程序设计思想与数据操作方法融会贯通，内容由浅入深

第五讲–学习内容

- 5.1递归函数
- 5.2模块化程序设计方法

5.1.1递归函数-函数的嵌套调用

- C语言规定函数不能嵌套定义
  - 函数是相互平行的，该限制可以使编译器简单化
- 但可以嵌套调用
  - 在调用一个函数的过程中又调用另一个函数



函数直接或间接调用自己，称为递归调用（Recursive Call），这样的函数，称为递归函数（Recursive Function）

5.1.1递归函数-递归调用

【例】计算n！ = n *(n-1)*(n-2)*……*1

```
long Fact(int n)
{
    if (n < 0)
        return -1;
    else if (n == 0 || n == 1)
        return  1;
    else
        return n * Fact(n-1);
}
```

递归调用，这样的函数，称为递归函数

【例5.1】计算n!

$$n!=\begin{cases}1 & n=0,1\\ n\times(n-1)! & n\geqslant 2\end{cases}$$

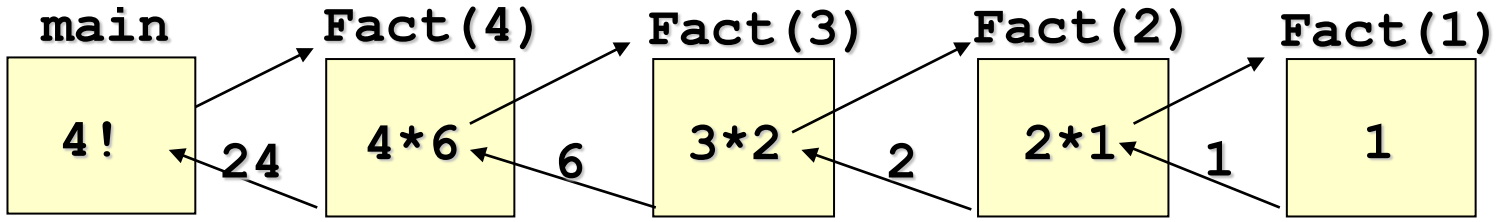
```
unsigned long Fact(unsigned int n)
  {
      if (n == 0 || n == 1)
          return 1;
      else
          return n * Fact(n-1);
  }
```

基本条件(base case):

一般条件
(general case)

无需考虑
n<0了

5. 1. 3递归函数-递归的计算过程



Fact(4) = 4 * Fact(3)
= 4 * (3 * Fact(2))
= 4 * (3 * (2 * Fact(1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24

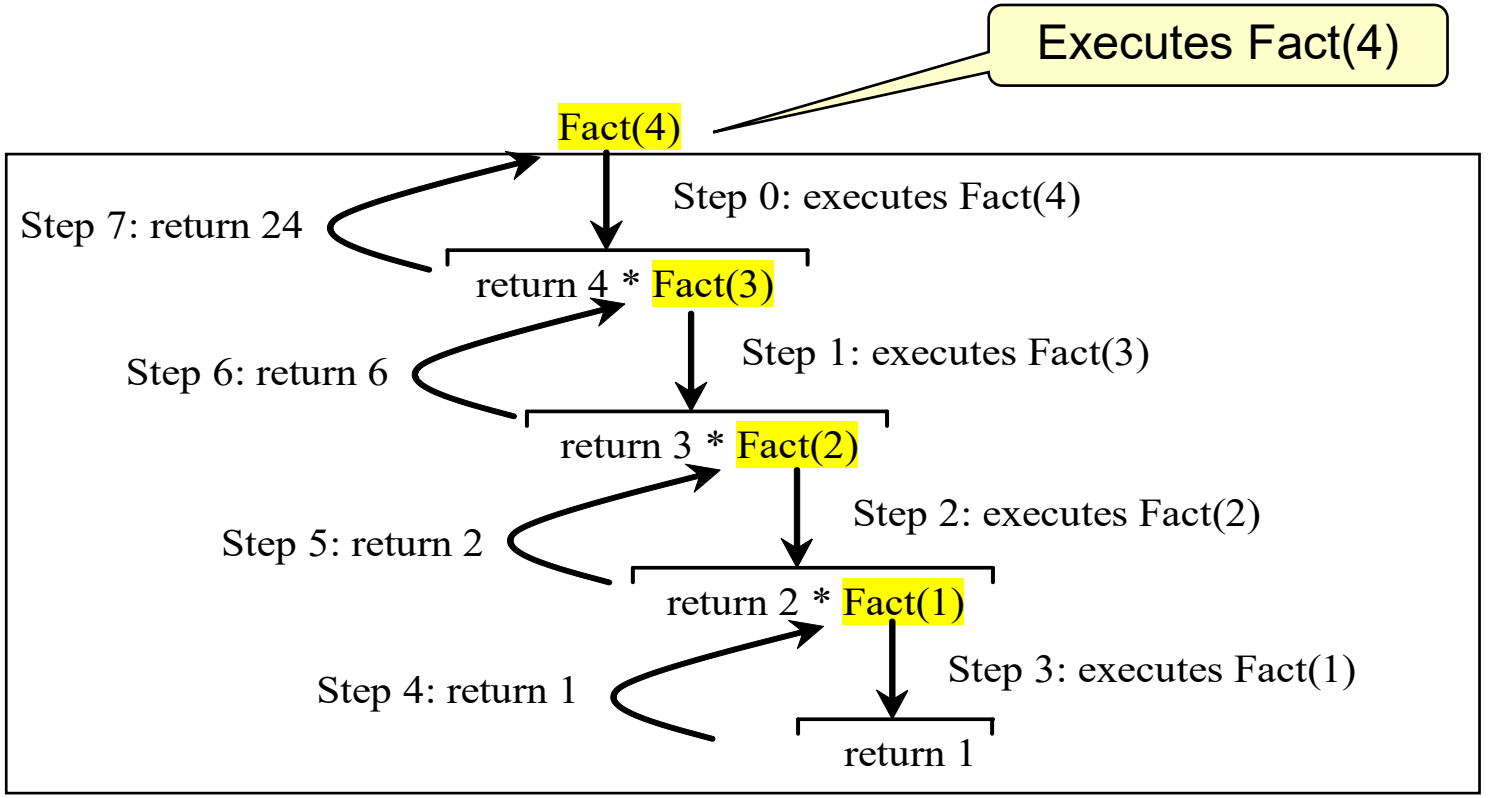
Fact(1) = 1;

Fact(n) = n * Fact(n-1);

复杂问题逐步简化，最终转化为一个最简单的问题

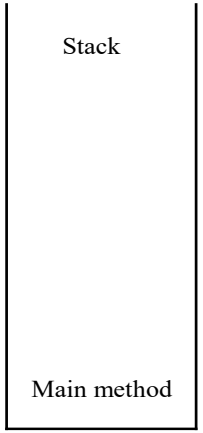
最简单问题的解决意味着整个问题的解决

5. 1. 3递归函数-递归的计算过程

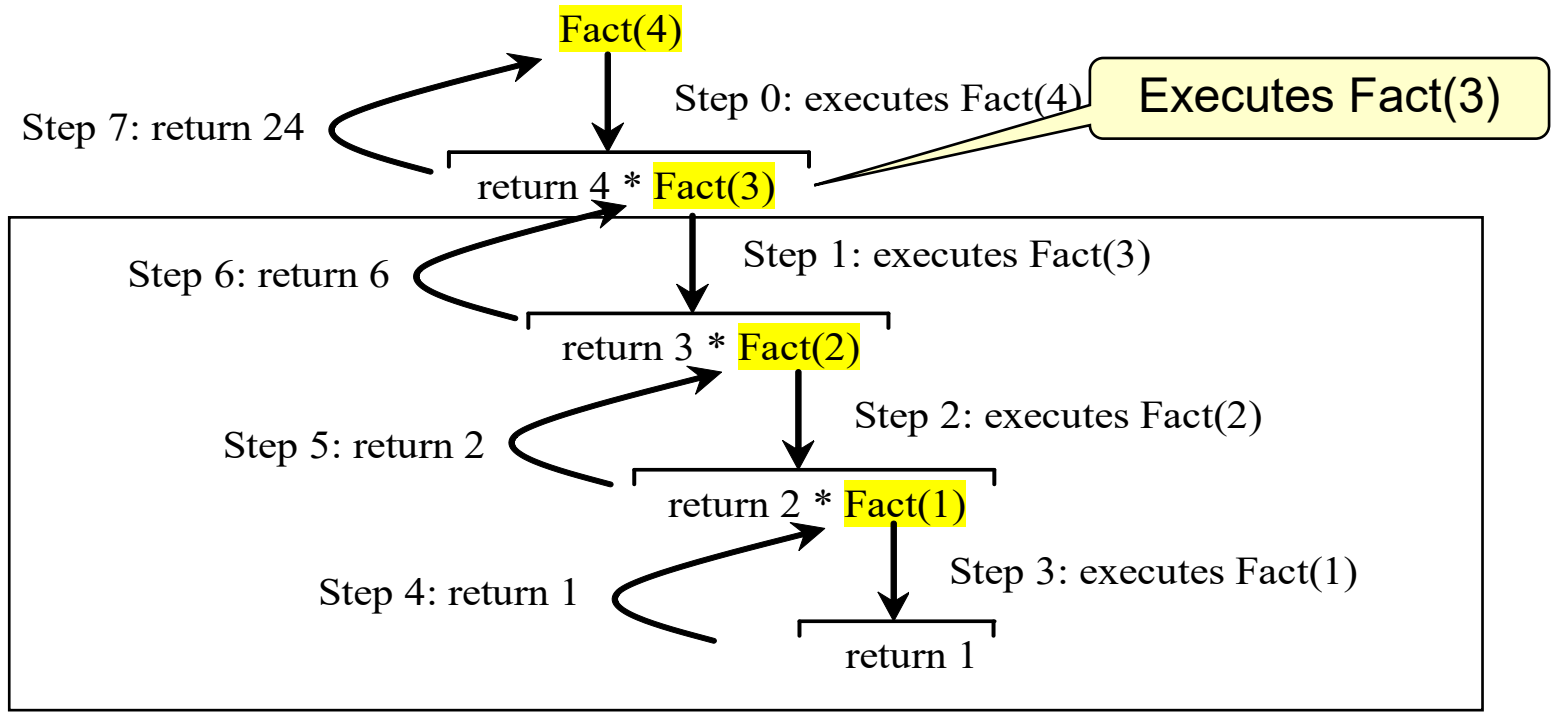


回推过程

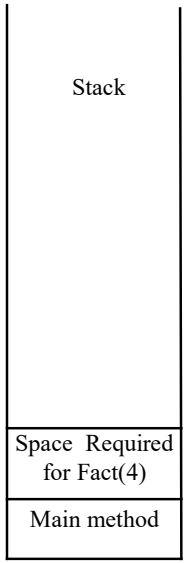
函数调用栈（Call stack）



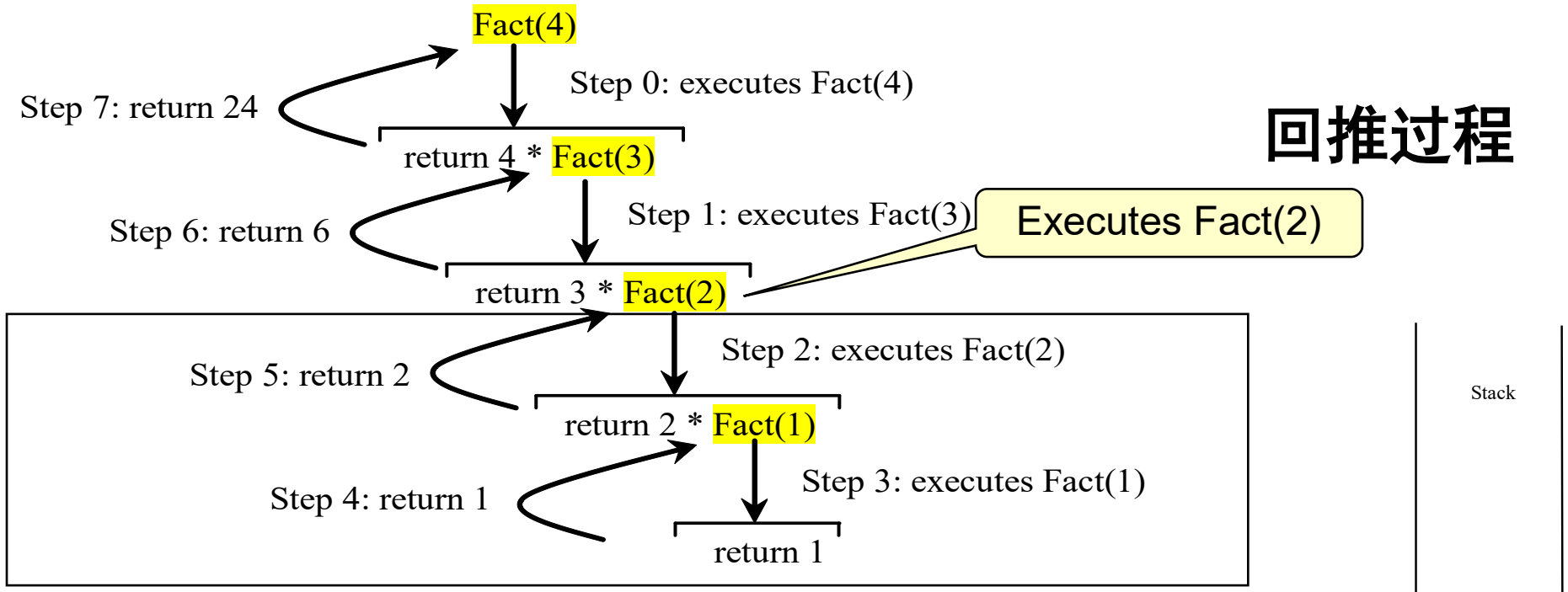
5. 1. 3递归函数-递归的计算过程



回推过程

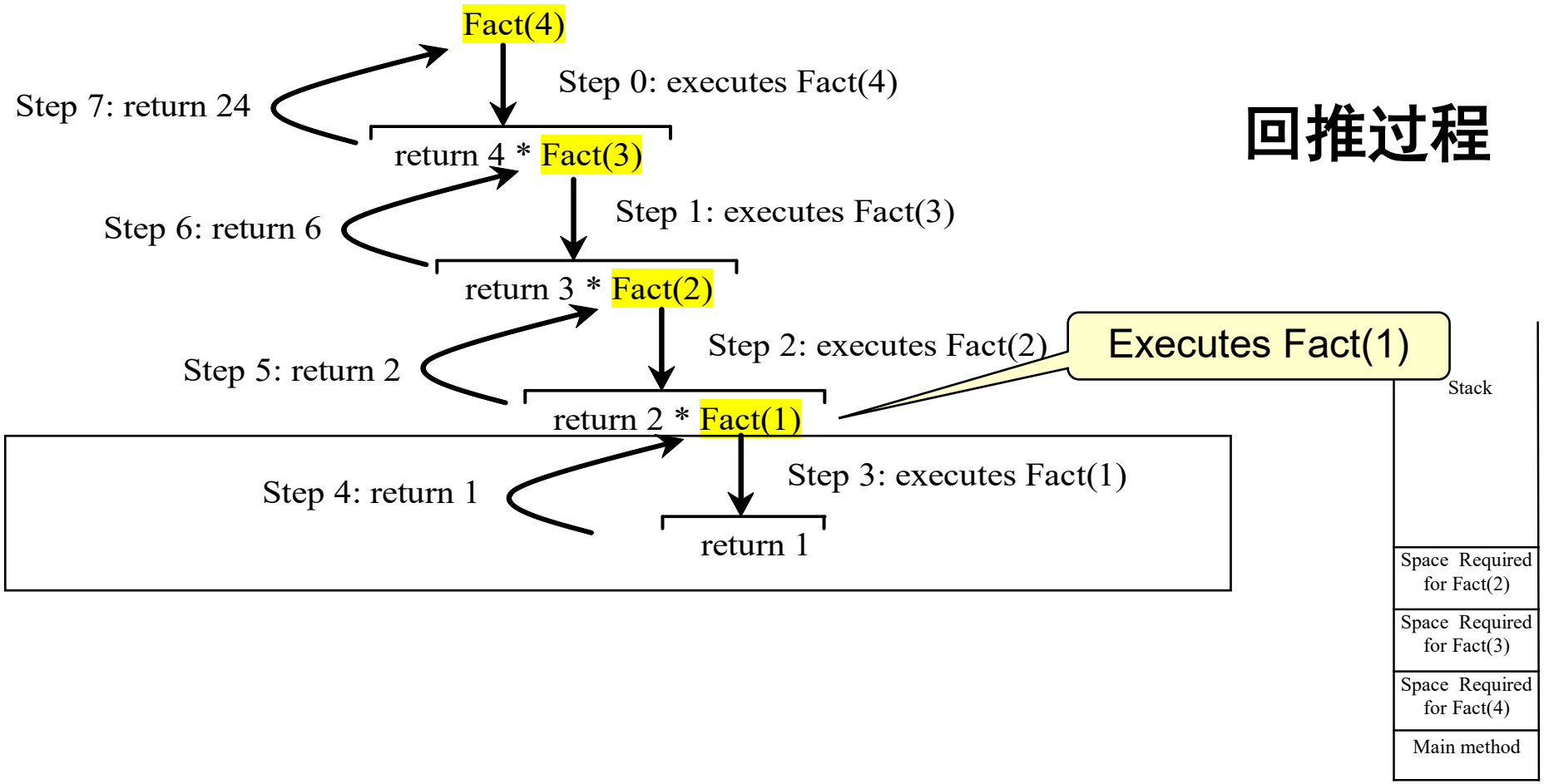


5. 1. 3递归函数-递归的计算过程



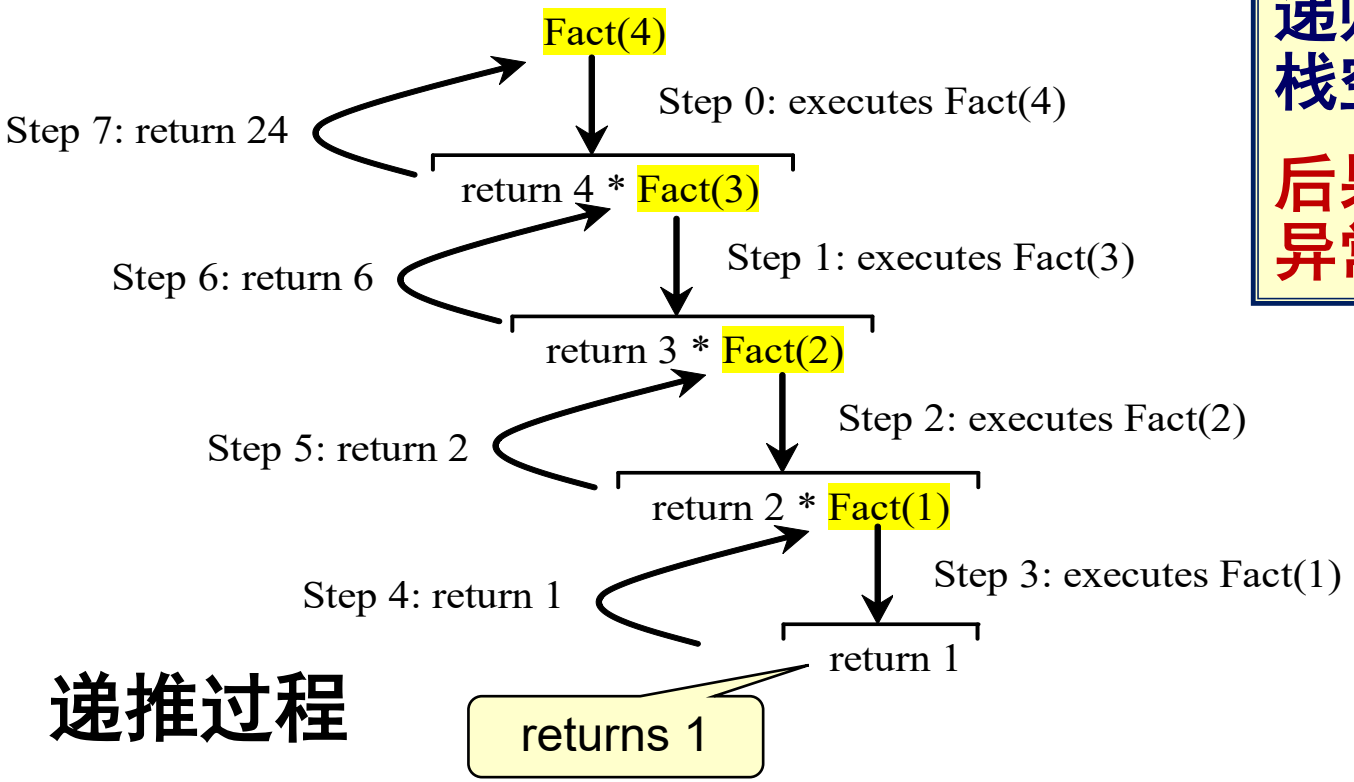
回推过程

5. 1. 3递归函数-递归的计算过程



回推过程

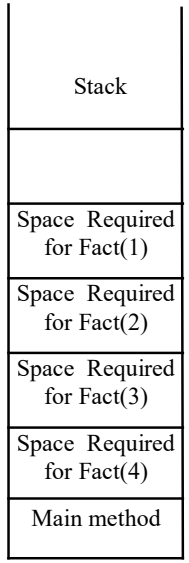
5. 1. 3递归函数-递归的计算过程



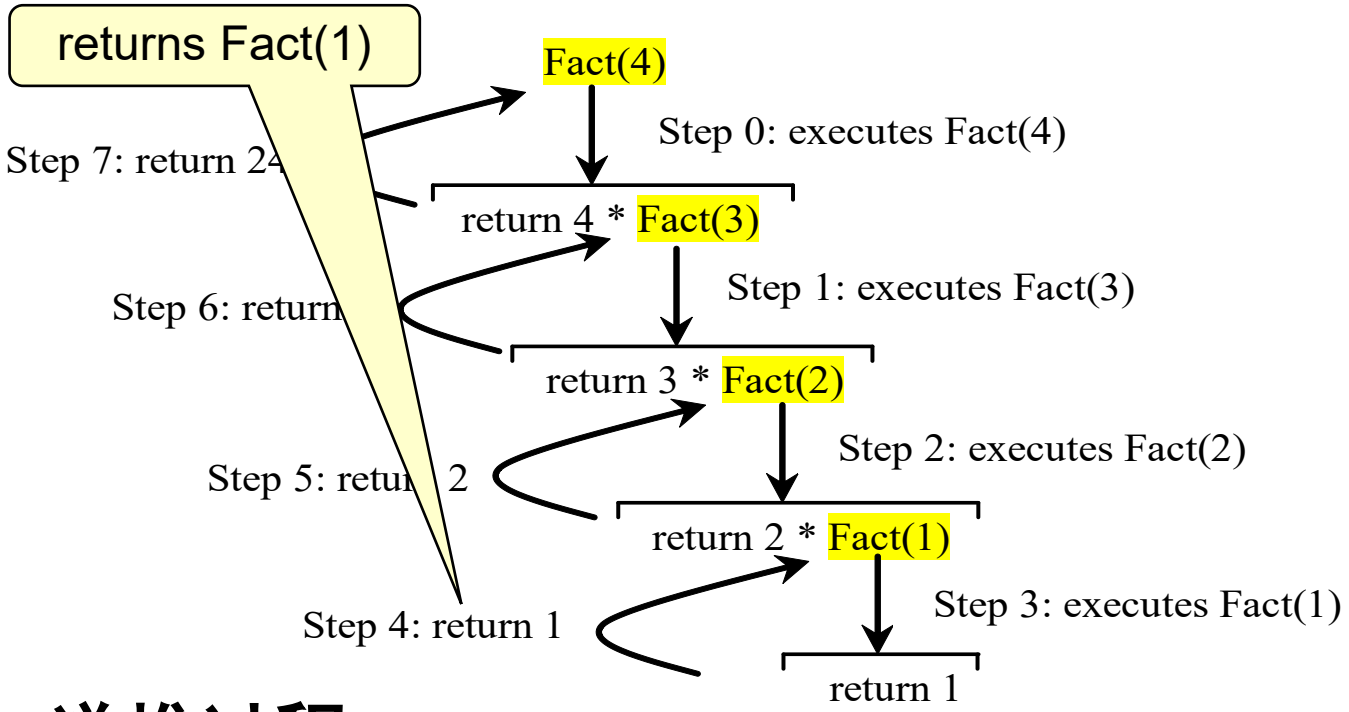
递推过程

递归层数太多易导致栈空间溢出

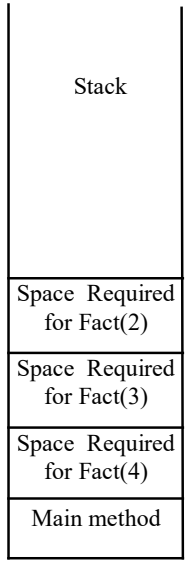
后果很严重，程序被异常中止



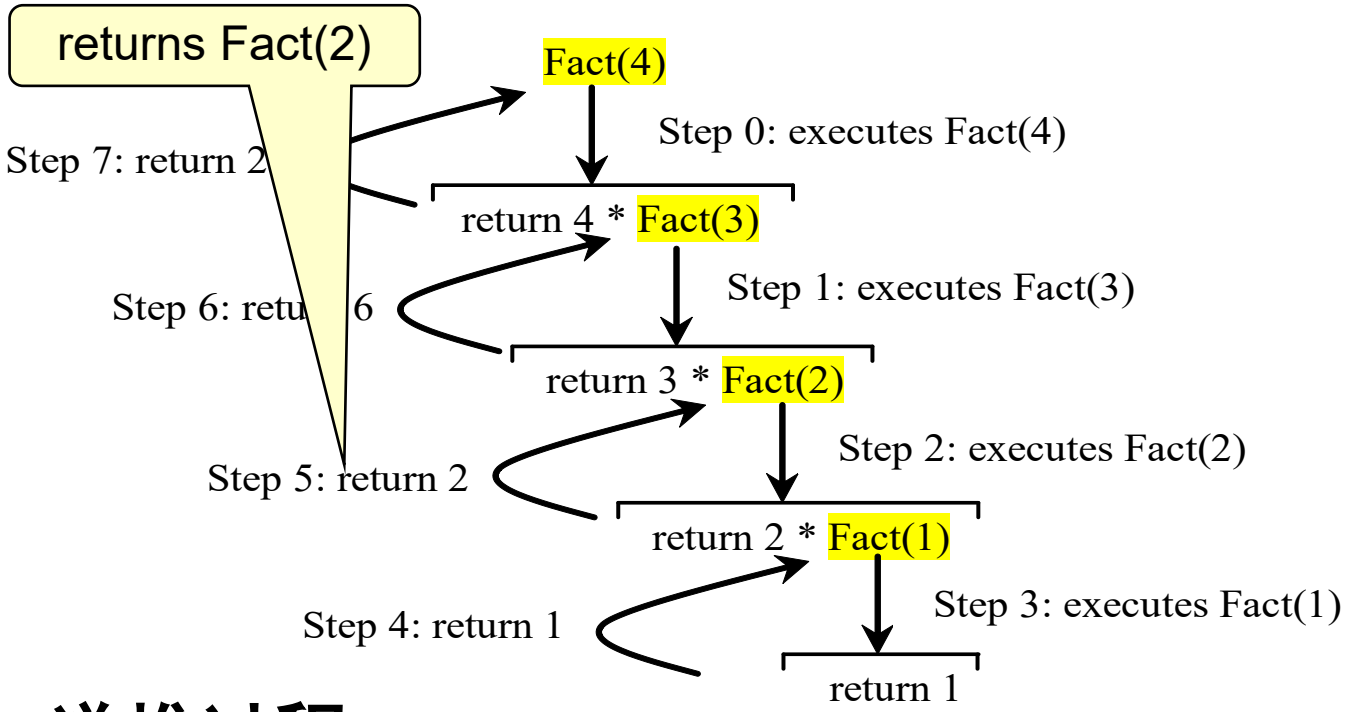
5. 1. 3递归函数-递归的计算过程



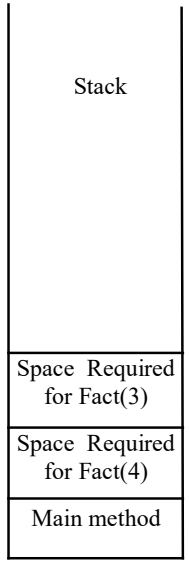
递推过程



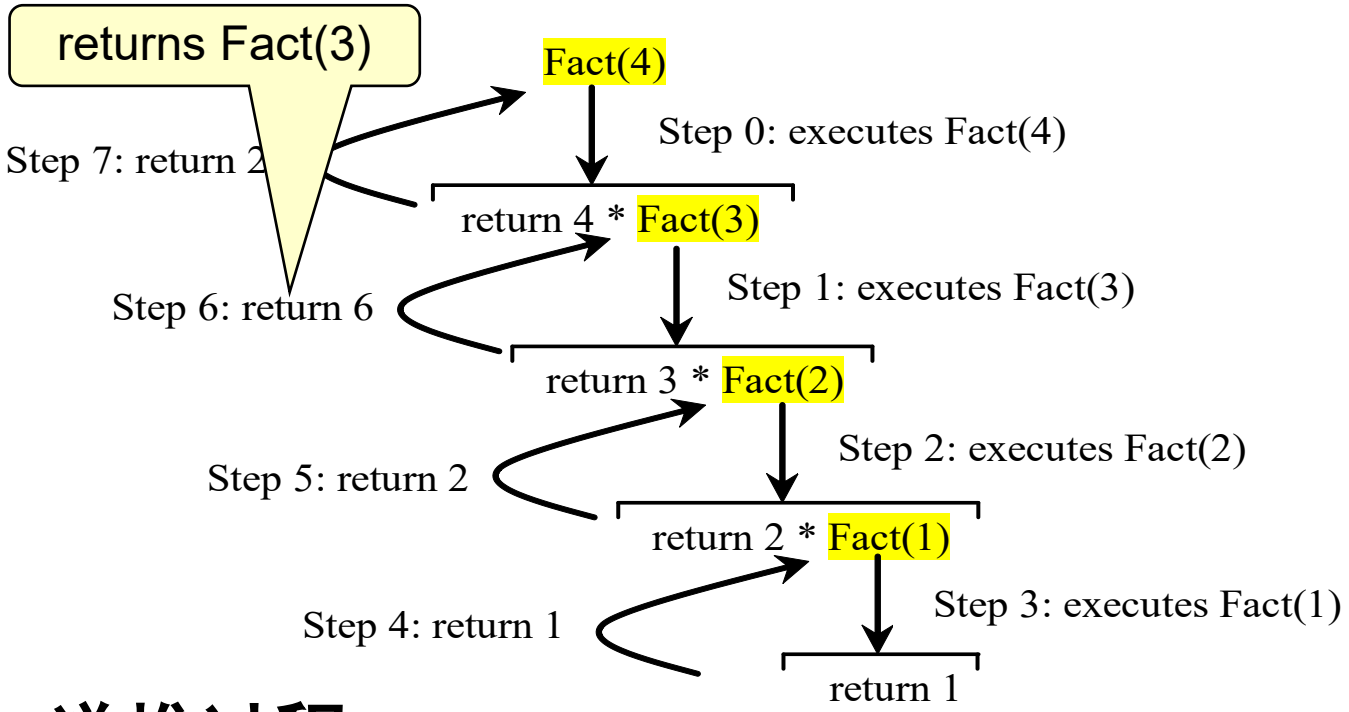
5. 1. 3递归函数-递归的计算过程



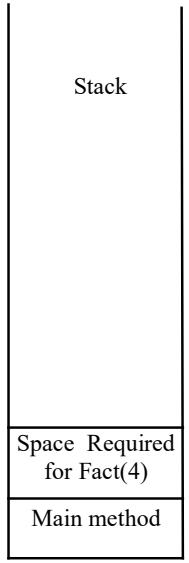
递推过程



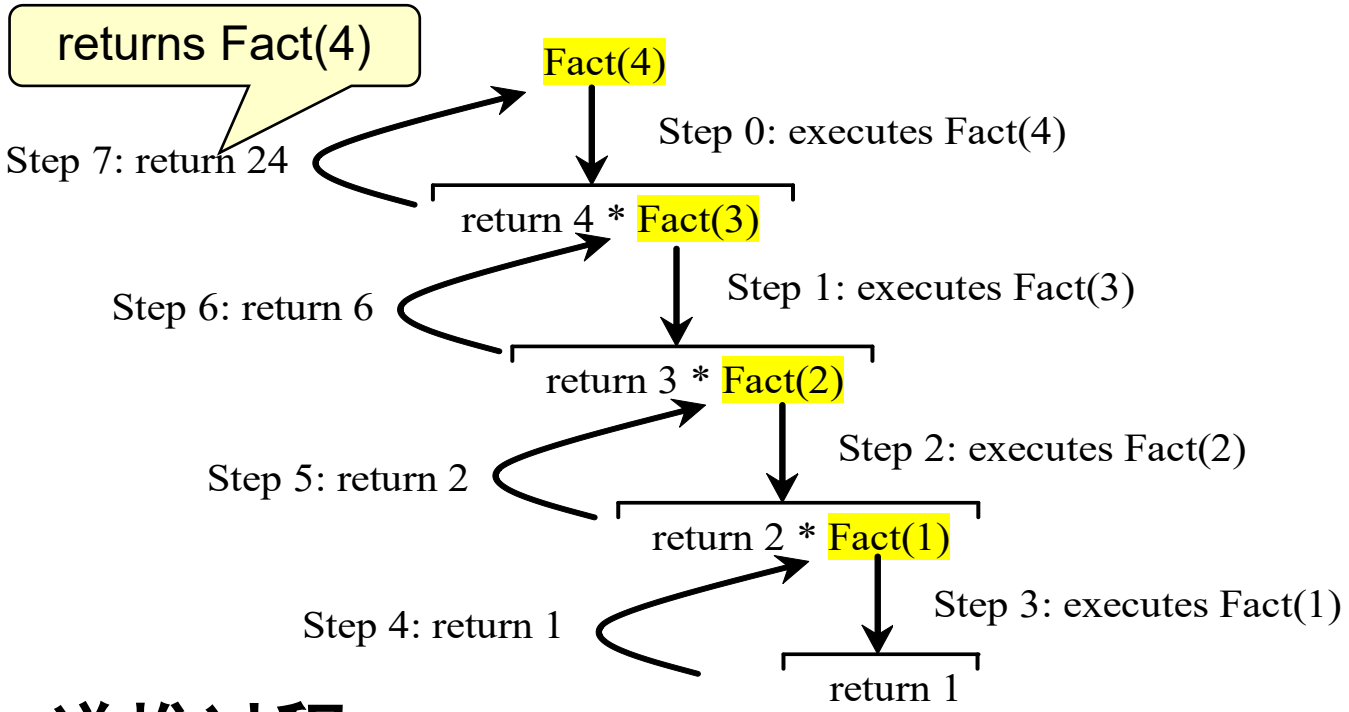
5. 1. 3递归函数-递归的计算过程



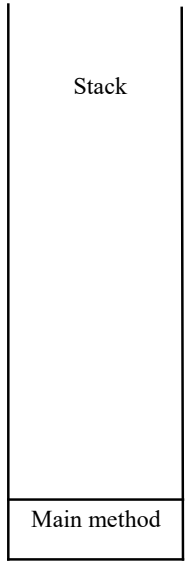
递推过程



5. 1. 3递归函数-递归的计算过程



递推过程



函数调用栈（Call stack）

5.1.2递归函数-递归函数的控制流程

- 任何一个递归调用程序必须包括两部分

仅当满足一定条件时，递归终止，称为条件递归

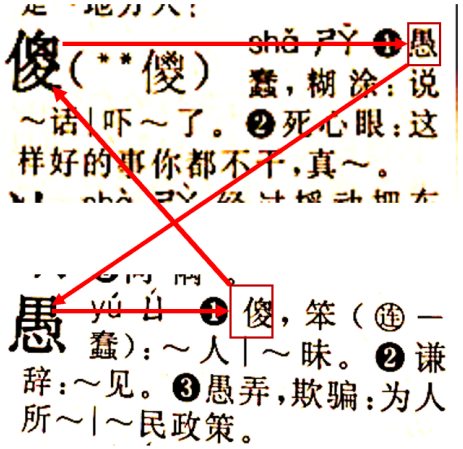
```
if (递归终止条件)
    return   递归公式的初值;
else
    return   递归函数调用返回的结果值;
```

基本条件控制递归调用结束

一般条件控制递归调用向基本条件转化

5.1.2递归函数-递归函数的控制流程

- 任何递归调用都必须向着“基本条件”的方向进行
  - 递归调用能在有限次数内终止，否则将无限循环
- 避免无穷递归
  - 老和尚讲故事的“无穷故事”
    - 从前有座山，山上有座庙，庙里有个老和尚，老和尚给小和尚讲故事，故事说：从前有座山……
  - 字典里的循环定义



5.1.2递归函数-优点分析

- 递归方法编写程序的优点
  - 符合人的思维习惯，逼近数学公式的表示
    - 递归求阶乘程序遵循了数学中对阶乘的定义
  - 从编程角度来看，简洁、直观、精炼，易编、易懂、逻辑清楚，结构清晰、可读性好

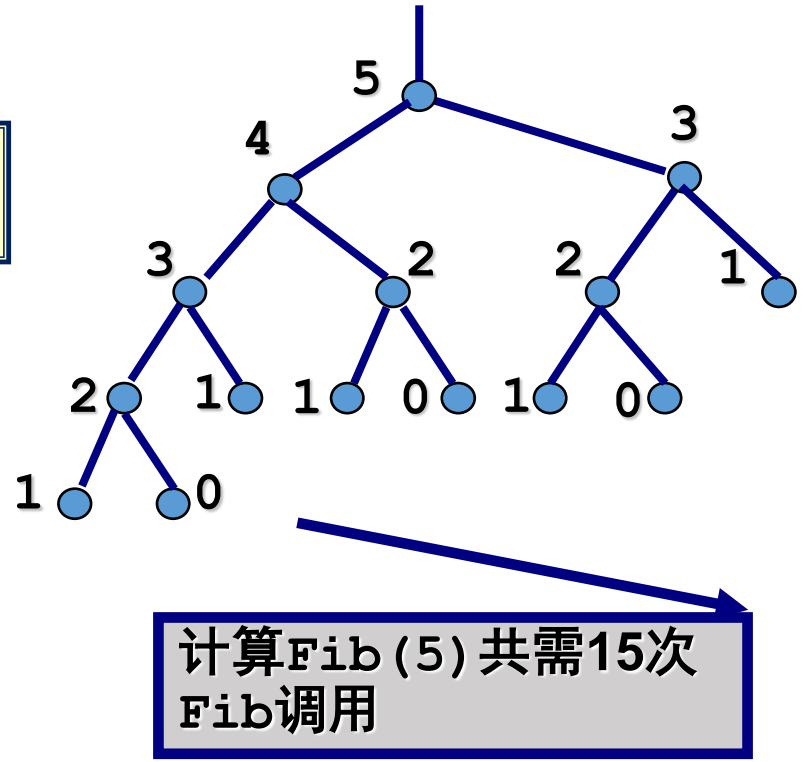
【例5.2】计算斐波那契数列

0，1，1，2，3，5，8，……

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

```
long Fib(int n)
{
  long f;
  if (n == 0) f = 0;
  else if (n == 1) f = 1;
  else f = Fib(n-1) + Fib(n-2);
  return f;
}
```

应尽量用迭代形式替代递归形式



计算Fib(5)共需15次Fib调用

5.1.2递归函数-缺点

- 递归方法编写程序的缺点
  - 增加了函数调用开销，每次调用都需参数传递、现场保护等，为函数使用的参数、局部变量等额外分配存储空间
  - 耗费更多的时间和栈空间，时空效率低
  - 重复计算多

5. 1. 4递归函数-使用场景

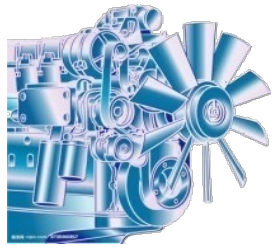
- 通常下面三种情况需要使用递归:
  - 数学定义递归的，如计算阶乘
  - 数据结构是递归的，如单向链表
  - 问题的解法是递归的
    - 非数值计算领域存在很多必须用递归法才能解决的经典问题
      - Hanoi塔，骑士游历、八皇后问题（回溯法）

5.2.1 模块化程序设计-模块化

- 模块各司其职
  - 每个模块只负责一件事情，它可以更专心
  - 一个模块一个模块地完成，最后再将它们集成
  - 便于单个模块的设计、开发、调试、测试和维护
- 开发人员各司其职
  - 按模块分配任务，职责明确
  - 并行开发，缩短开发时间

5.2.1 模块化程序设计-模块化

- 某一功能，若重复实现3遍以上则应考虑模块化，将它写成通用函数，向小组成员发布
- 拿来拿去主义，不是人类懒惰的表现，而是智慧的表现

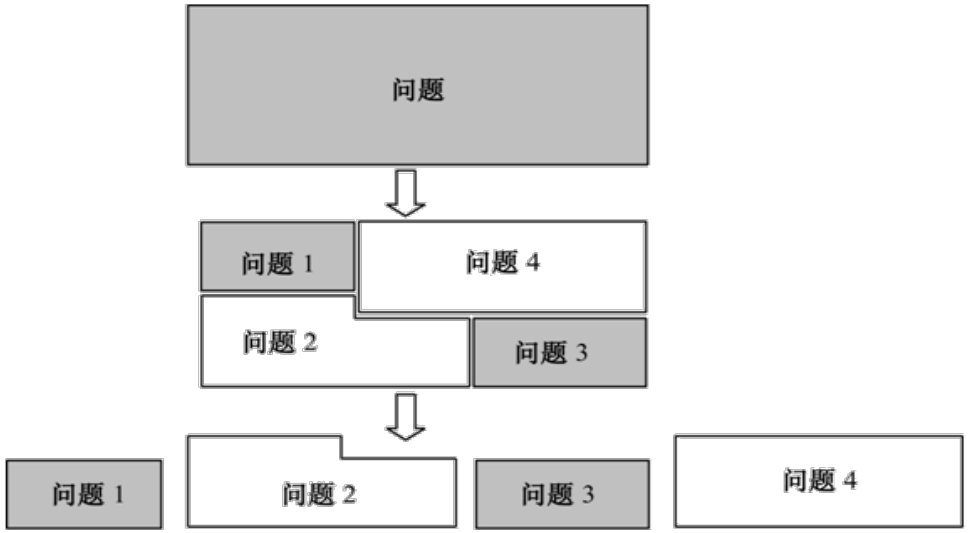


5.2.1 模块化程序设计-模块化

- 模块化的优点——便于复用
  - 构建新的软件系统可以不必每次从零做起
  - 直接使用已有的经过反复验证的软构件，组装或修改后成为新的系统
  - 提高软件生产效率和程序质量

5.2.1 模块化程序设计-模块化

- 模块分解的过程
  - 自顶向下（Top-down）、逐步求精（Stepwise refinement）的程序设计方法
    - 先全局后局部，先整体后细节，先抽象后具体
  - 由不断的自底向上修正所补充的自顶向下的程序设计方法



5.2.3 模块化程序设计-模块分解与模块接口

- 模块分解的基本原则
  - 保证模块的相对独立性——高聚合、低耦合
  - 模块的实现细节对外不可见——信息隐藏
    - 外部：关心做什么；内部：关心怎么做
- 模块接口设计
  - 指罗列出一个模块的所有与外部打交道的变量
  - 定义好后不要轻易改动
  - 在模块开头（文件的开头）进行函数声明

5.2.3 模块化程序设计-预编译指令

- 预编译不是编译器的组成部分，但是它是编译过程中一个单独的步骤，是一个文本替换工具，它们会指示编译器在实际编译之前完成所需的预处理。
- 以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。
- **预编译指令举例：**
  - #include <stdio.h>
  - #define MAX 100

5.2.3 模块化程序设计-预编译指令

- **1.文件包含: #include**
  - 是一种最为常见的预处理，主要是做为文件的引用组合源程序正文。
  - #include <stdio.h>
  - #include <math.h>
  - #include <stdlib.h>
- **2.条件编译:#if, #ifndef, #ifdef, #endif, #undef**
  - 主要是进行编译时进行有选择的挑选，注释掉一些指定的代码，以达到版本控制、防止对文件重复包含的功能。
  - #ifndef MYFILE_H
  - #define MYFILE_H
  - #endif

5.2.3 模块化程序设计-预编译指令

- **3.布局控制: #progma**
  - 主要功能是为编译程序提供非常规的控制流信息

- **4.宏替换: #define**
  - 这是最常见的用法，它可以定义符号常量、函数功能、重新命名、字符串的拼接等各种功能
  - #define MAX 100
  - #define SQUARE(x) ((x)*(x))

5.2.4 模块化程序设计-提高程序可维护性

- **1.模块化设计:**
  - 将实现特定功能的代码封装成函数或模块，这样当需要使用该功能时，可以直接调用，而不需要重复编写代码。这有助于减少代码几余，提高代码的复用性。
- **2.清晰的命名:**
  - 变量、函数和模块的命名应该清晰、准确，能够直观地反映其作用和用途。避免使用模糊不清或者过于简化的命名，这样可以大大提高代码的可读性。

```
int f(int a, int b)
{
    return a+b;
}

int Add(int a, int b)
{
    return a+b;
}
```

```
int n=5;
int book_num=5;
```

5.2.4 模块化程序设计-提高程序可维护性

- **3.注释和文档:**
  - 在代码中添加必要的注释，特别是对于复杂的逻辑和算法，以及公共接口的使用说明。同时，维护良好的文档，包括设计文档Q、用户手册等，有助于其他开发者快速理解和使用代码。
- **4.代码风格统一:**
  - 遵循一致的编码风格和规范，比如缩进、空格、括号的使用等，使得代码看起来整洁有序，便于阅读和维护.

5.2.4 模块化程序设计-提高程序可维护性

```c
#include <stdio.h>
main(){int fahr,celsius;
int lower,upper,step;
lower=0;upper=300;step=20;fahr=lower;
while (fahr<=upper){
celsius=5*(fahr-32)/9;
printf("%d\t%d\n",fahr,celsius);
fahr=fahr+step;}}
```

VS

```c
#include <stdio.h>
/* 对 fahr = 0, 20, ..., 300
   打印华氏温度与摄氏温度对照表 */
main()
{
    int  fahr, celsius;
    int  lower, upper, step;

    lower = 0;      /* 温度表的下限 */
    upper = 300;    /* 温度表的上限 */
    step  = 20;     /* 步长 */
    fahr  = lower;

    while (fahr <= upper)
    {
        celsius = 5 * (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

5.2.4 模块化程序设计-提高程序可维护性

- **5.控制结构简洁:**
  - 尽量使用简单的控制结构，避免过深的嵌套和复杂的条件判断，这样可以减少理解和维护的难度。

- **6、错误处理:**
  - 合理地外理错误情况，不要忽略异常，确保程序在各种情况下都能给出明确的错误信息或者采取恰当的措施

```
if()
    if()
        if()
            if()
                if()
                    if()
                        n=1;
                    else
                        n=2;
                else
                    n=3
            else
                n=4;
        else
            n=5;
    else
        n=6;
else
    n=7;
```

5.2.4 模块化程序设计-提高程序可维护性

- **7.代码审查:**
  - 定期进行代码审查，可以帮助发现潜在的问题和改进代码质量，同时也是知识共享和团队协作的好方法。
- **8.测试:**
  - 编写充分的测试用例，进行单元测试、集成测试和系统测试，确保代码的正确性和稳定性。
- **9.设计模式:**
  - 在适当的场合使用设计模式，可以提高代码的灵活性和扩展性，同时也使得代码更加易于理解和维护

5.2.4 模块化程序设计-提高程序可维护性

- **10.架构设计:**
  - 站在软件整体架构的角度去思考，合理规划软件的结构，使得各个部分职责分明，相互协作，提高整体的可维护性
- **11.避免过度工程:**
  - 即不要为了追求完美而过度设计和编写不必要的代码，这样会导致代码复杂难以维护
- **12.持续重构:**
  - 随着需求的变更和技术的进步，不断重构代码，去除不再需要的代码，优化现有的设计，保持代码的活力.

本讲小结

- **递归函数**
  - 递归的一般条件和基本条件 P10
  - 递归函数的计算过程 P11
- **模块化程序设计思想**
  - 模块化思想
  - 预编译指令 P56
  - 提高程序可维护性的方法 P59