北京郵電大學

第五章 函数--递归

北京郵電大學

提纲

1. 递归的概念
2. 递归过程
3. 递归程序设计

1.递归的概念

- 递归算法在可计算性理论中占有重要地位，它是算法设计的有力工具，对于拓展编程思路非常有用。就递归算法而言并不涉及高深数学知识，只不过初学者要建立起递归概念不十分容易。
- 我们先从一个最简单的例子导入。

1.递归的概念

例：编写一个函数fac，计算阶乘n!

按过去的迭代算法，该函数可以写成：

```
int fac(int n)
{
    int i, p;
    p = 1;
    for(i = 2; i <= n; i++)
        p = p * i;
    return p;
}
```

1.递归的概念

现在换一个角度考虑，n！不仅是1×2×3×…×n，还可以定义成：

$$n! = \begin{cases} 1 & \text{当} n=0 \\ n\times(n-1)! & \text{当} n>0 \end{cases}$$

```
int f(int n)
{
    if(n == 0)
        return 1;
    else
        return n * f(n-1);
}
```

设 f(n)=n!

$$\text{则 } f(n) = \begin{cases} 1 & \text{当} n=0 \\ n\times f(n-1) & \text{当} n>0 \end{cases}$$

根据以上数学定义，函数f能
左边所示？函数能否调用自

答案是肯定的！C系统会保证调用过程的正确性，这就是递归！

递归的定义：

- 从程序书写来看，在定义一个函数时，若在函数的功能实现部分又出现对它本身的调用，则称该函数是递归的或递归定义的。
- 从函数动态运行来看，当调用一个函数A时，在进入函数A且还没有退出（返回）之前，又再一次由于调用A本身而进入函数A，则称之为函数A的递归调用。

```
int f(int n)
{
    if(n == 0)
        return 1;
    else
        return n * f(n-1);
}
```

1.递归的概念

递归可以分为直接递归和间接递归两种。

直接递归：函数体里面发生对自己的调用；

间接递归：函数A调用函数B，而函数B又直接或间接地调用函数A。

直接递归

```
A(…)
{
    …
    A(…);
    …
}
```

间接递归

```
A(…)
{
    …
    B(…);
    …
}
```

```
B(…)
{
    …
    A(…);
    …
}
```

1.递归的概念

- 不用担心函数A内部又调用函数A，会使得调用无休无止，肯定存在某个条件，当该条件成立的时候，函数A将不会再调用自身。

例如，求n！时，该结束条件是 if(n == 0)

```
int f(int n)
{
   if(n == 0)
      return 1;
   else
      return n * f(n-1);
}
```

提纲

1. 递归的概念
2. 递归过程 （递归计算过程）
3. 递归程序设计

过程

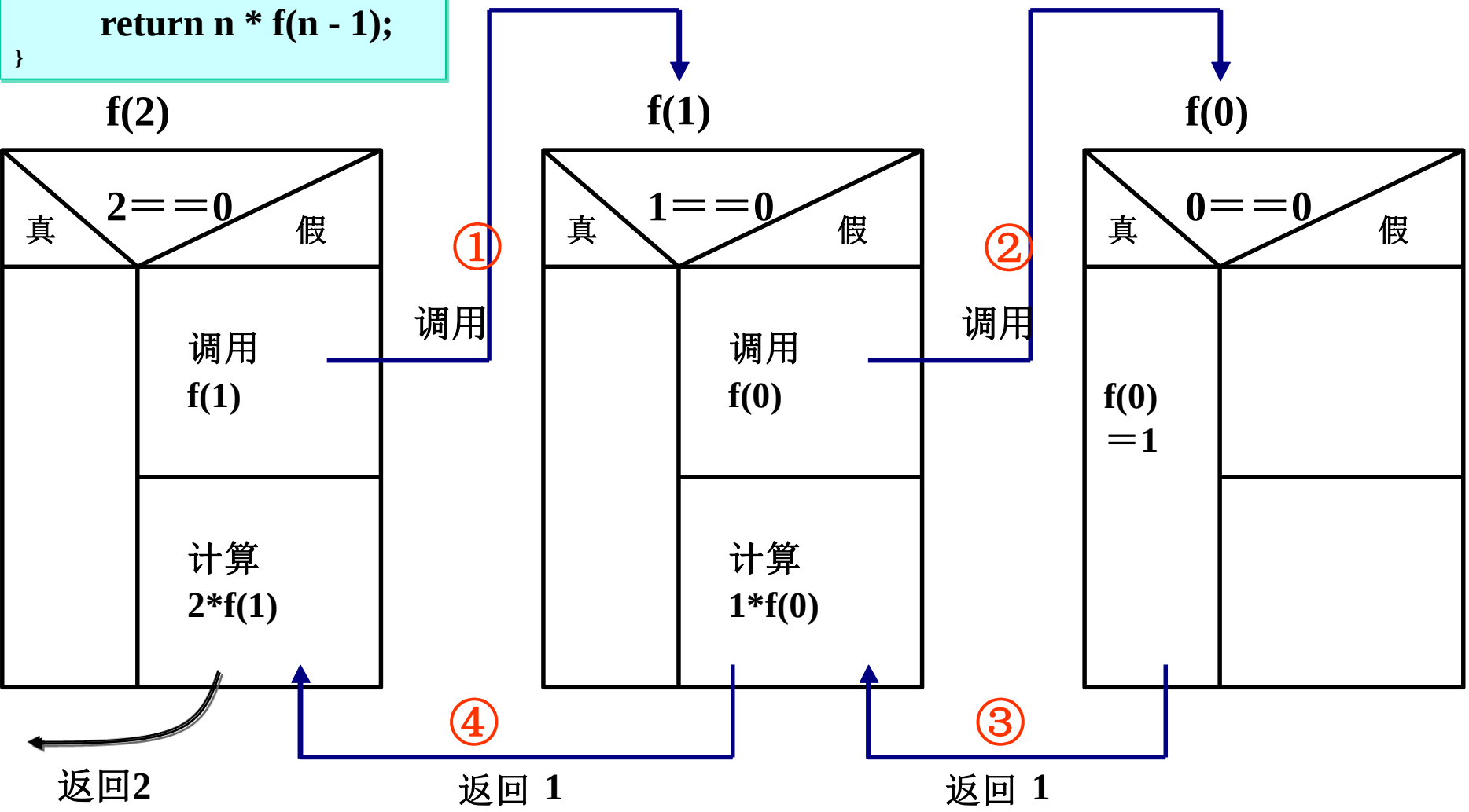
- 求f（2）的递归调用过程？

需要关注递归调用是如何计算结果?

```
#include <stdio.h>
int f(int);
main()
{
    int i = 2;
    printf("%d",f(i));
    system("pause");
}
int f(int n)//递归函数：求n!
{
    if(n == 0)
        return 1;
    else
        return n * f(n - 1);
}
```

```
int f(int n)
{
    if(n == 0)
        return 1;
    else
        return n * f(n - 1);
}
```

2.递归过程



2.递归过程

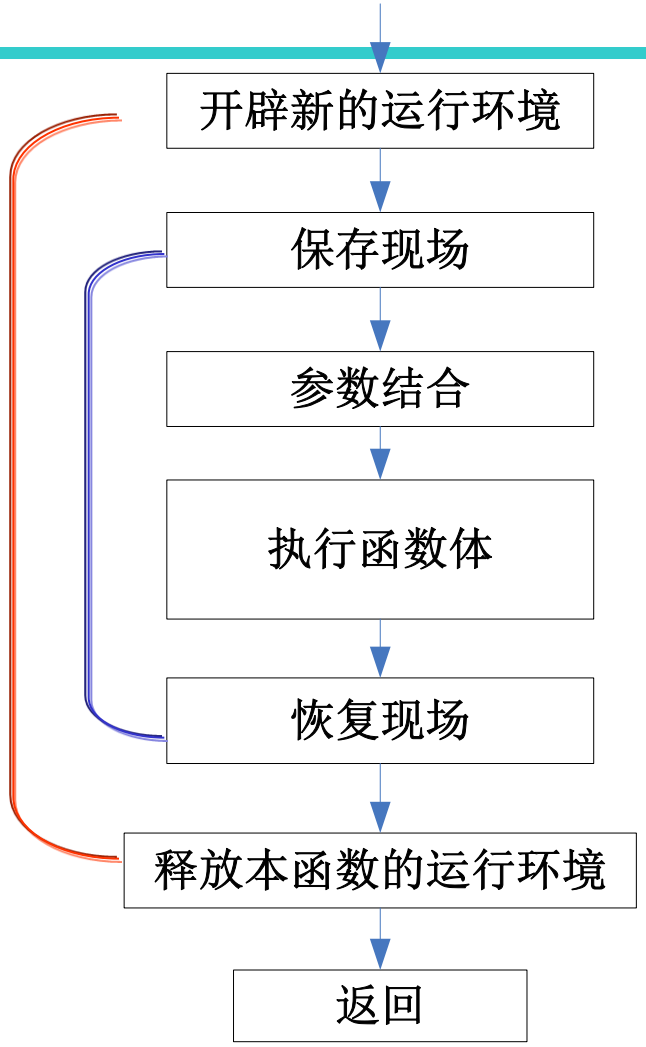
- 请思考：
  - 发出f(2)调用时，将2赋值给形参n。然后发出f(1)调用，将1赋值给形参n。接着发出f(0)调用，将0赋值给形参n。后来赋给形参n的值会不会覆盖原来赋给n的值（如值1覆盖原来的值2）？为什么？

    —不会，每一次函数调用会在栈顶分配新的活动记录。
  - 对递归函数的每一次调用结束返回时，为何能回到调用前的程序运行状态？如f(1)调用结束返回f(2)时，n的值为2。

    —函数访问的永远是栈顶的活动记录。当f(1)调用结束时，位于栈顶的f(1)的活动记录将出栈，此时位于栈顶的将是f(2)的函数活动记录。

2.递归过程

回顾



函数调用过程

被调用函数中的数据

现场信息，用于调用结束能正确返回并执行

局部变量
形参(放实参的值)
机器的状态信息
返回地址

简化的函数活动记录

```
5.printf("%d",f(i));
8.int f(int n)
9.{
10.    if(n==0)
11.       return 1;
12.    else
13.       return n*f(n-1);
14.}
```

```
1.  int f(int);
2.  main()
3.  {
4.      int i=2;
5.      printf("%d",f(i));
6.      system("pause");
7.  }
8.  int f(int n)//递归函数：求n!
9.  {
10.     if(n==0)
11.        return 1;
12.     else
13.        return n*f(n-1);
14. }
```

2
操作系统

步骤（2）

调用f(0)

n	0
返址	/*13*/
n	1
返址	/*13*/
n	2
返址	/*5*/
i	2
返址	操作系统

步骤（5）

f(0)返回值1

n	1
返址	/*13*/
n	2
返址	/*5*/
i	2
返址	操作系统

步骤（6）

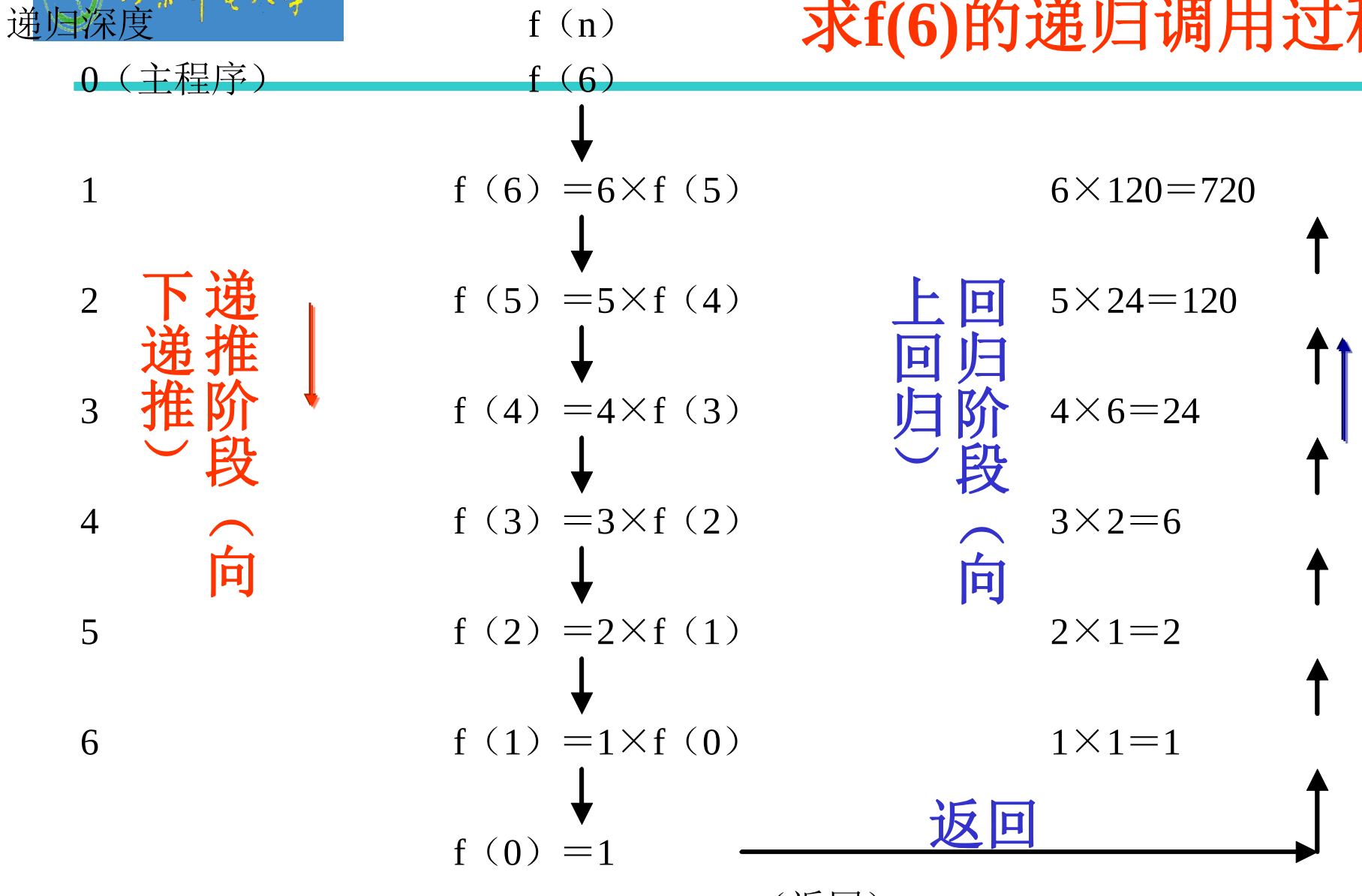
i	2
返址	操作系统

步骤（7）

i	2
返址	操作系统

步骤（8）

求f(6)的递归调用过程


递归深度
f（n）
0（主程序）
f（6）
1
f（6）＝6×f（5）
2
f（5）＝5×f（4）
3
f（4）＝4×f（3）
4
f（3）＝3×f（2）
5
f（2）＝2×f（1）
6
f（1）＝1×f（0）
f（0）＝1
递推阶段（向下递推）
回归阶段（向上回归）
6×120＝720
5×24＝120
4×6＝24
3×2＝6
2×1＝2
1×1＝1
返回
（返回）
[递推阶段]
[回归阶段]

2.递归过程

- 可见，递归算法的执行过程分递推和回归两个阶段。当递推时，并没有求值的计算操作，实际的计算操作是在回归过程实现的。
- 递推阶段：
  - 递推阶段是个不断简化问题的阶段：把对较复杂问题（规模为n）的求解转化为比原问题简单一些的问题（规模小于n）的求解 。例如对f (6)的求解转化为f(5)的求解， 对f(5)的求解转化为f(4)的求解…直到转化为对f(0)的求解。
  - 当递推到最简单的不用再简化的问题时，递推终止。如f函数中，n==0的情况。
  - 就象剥一颗圆白菜，从外向里，一层层剥下来，到了菜心，就不再继续剥了。

2.递归过程

- 回归阶段：
  - 在回归阶段，当获得最简单情况的解后，逐级返回，依次得到稍复杂问题的解 。如在得到f(1)的解后，又依次得到f(2)、f(3)…直到f(6)的值。
  - 就像将剥下来的白菜叶又从里往外一叶一叶合起来，直到最外层菜叶。

2.递归过程

- 思考：递归与迭代的比较(以求阶乘为例)。
  - 迭代:从已知的初始条件出发，逐次去求所需要的阶乘值，这相当于从菜心“推到”（通过循环）最外层的菜叶。

$$f(0) = 1$$

$$f(1) = 1*f(0) = 1$$

$$f(2) = 2*f(1) = 2$$

  - 递归：先从最外层的菜叶“递推”到菜心(递归函数调用)，再从菜心“回归”到最外面的菜叶（递归函数返回，**带值返回**）。

2.递归过程

- 递归算法的出发点不放在初始条件上，而放在求解的目标上，是从所求的未知项出发逐次调用本身的求解过程，直到递归的边界（即初始条件）。
- 就求阶乘而言，读者会认为递归算法可能是多余的，费力而不讨好。但许多实际问题不可能或不容易找到显而易见的迭代关系，这时递归算法就表现出了明显的优越性。
- 下面我们将会看到，递归算法比较符合人的思维方式，逻辑性强，可将问题描述得简单扼要，具有良好的可读性，易于理解，许多看来相当复杂，或难以下手的问题，如果能够使用递归算法就会使问题变得易于处理。

北京郵電大學

提纲

1. 递归的概念
2. 递归过程
3. 递归程序设计

3.递归程序设计

- 什么样的问题可以用递归解决？

  如果解决问题的方法是把该问题分解成小的子问题，并且这些小的子问题可以用同样的算法解决，这样不断分解，直到子问题比较简单、可以直接解决时分解过程即终止，那么就可以用递归。

  递归的思想就是先将一个**问题转化**为与原问题性质相同、但规模小一级的子问题，然后再重复这样的转化，**直到问题的规模减小到我们很容易解决为止**。 -分治法（典型例子-称金块）

https://blog.csdn.net/weixin_72535480/article/details/128500576

3.递归程序设计

◆一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归逐级返回。

◆如何设计递归算法:

1）对所求解的问题、要计算的函数书写递归定义；注意一定要有终止条件和对应的操作。

2）正确地设计递归函数的参数。

注意：递归算法最外层肯定采用的是选择结构！为什么？

递归程序设计举例

练习1.求浮点数x的n次幂(n≧0)。函数 x^n 递归定义:

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

```
float power(float x，int n)
{
    if (n == 0)
        return 1;
    else
        return  x * power(x, n-1);
}
```

1）选择结构
2）自己调用自己肯定在选择结构某一分支
3）把f(n)=n*f(n) 右边的表达式写到return

递归程序设计举例

练习2. 设计递归函数求任意正整数的位数。num=1234

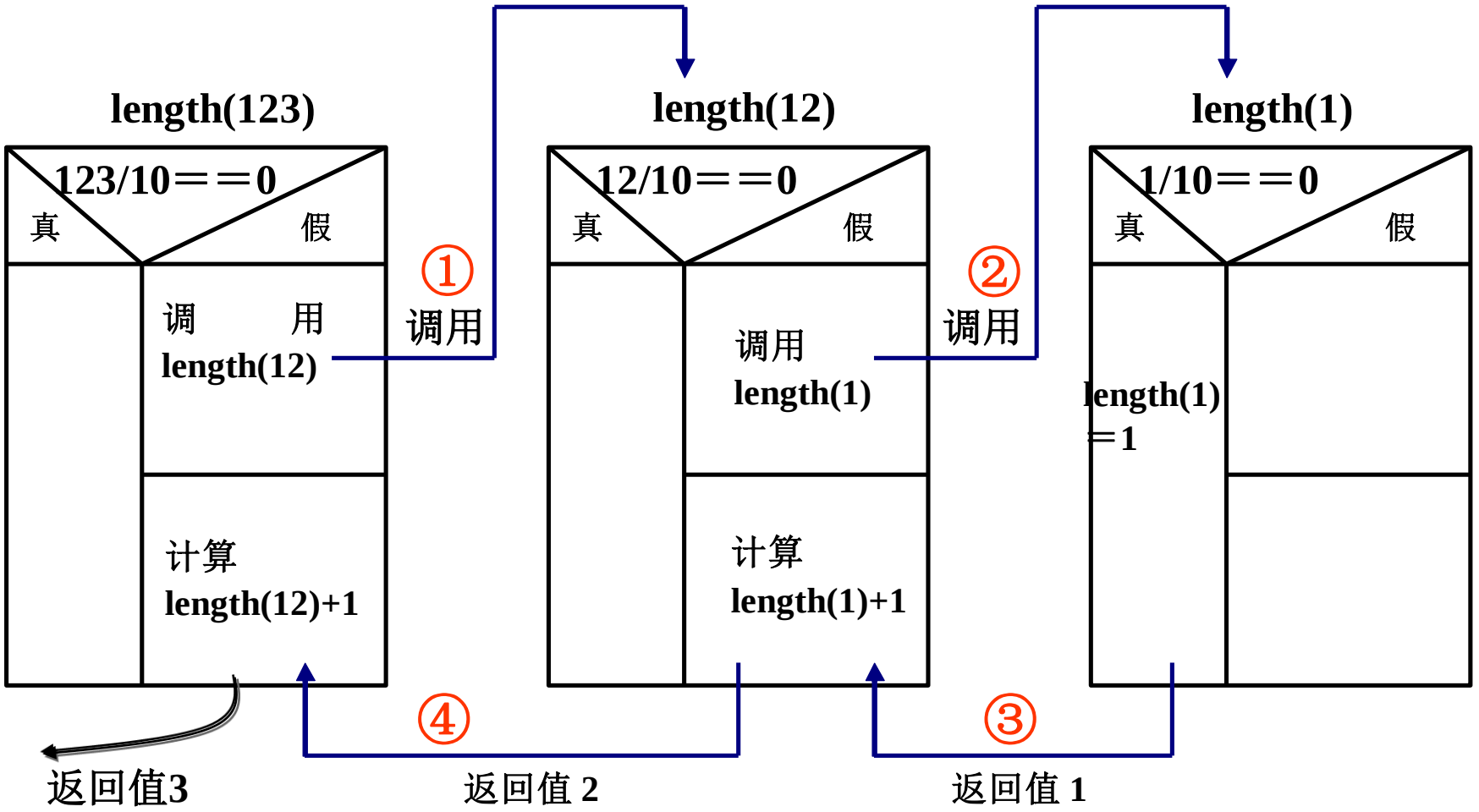
$$length(num)=\begin{cases}1 & \text{if } num/10=0\\ length(num/10)+1 & \text{if } num/10\neq 0\end{cases}$$

```
int len
{
  int len;
  len=0;
  if (num / 10 == 0)
      len=1;
  else //从最低位开始，依次从n中砍掉各位
     while (num > 0){
        num = num / 10;//去掉最低位
        len++; /*分解出一位数字，长度加1*/
     }

   return len;
}
```

```
int length (int num)
{
    if (num / 10 == 0)
        return 1;
    else
        return  length(num / 10) + 1;
}
```

递归程序设计举例


length(123)
123/10==0
真
假
调 用
length(12)
计算
length(12)+1
①
调用
length(12)
123/10==0
真
假
调用
length(1)
计算
length(1)+1
②
调用
length(1)
1/10==0
真
假
length(1)
=1
③
返回值 1
④
返回值 2
返回值3

递归程序设计举例

练习3.求任意正整数的逆置数-重点。 num=1234

递归思路1：将除最高位之外的数先逆置。

例如： reverse (1234)=1＋reverse(234）×10

思路1递归定义：

reverse (num)=

{ num　　if num长度为1

{ highBit+reverse(restNum)×10　　if num长度大于1

其中： highBit = num / power(10, len-1);

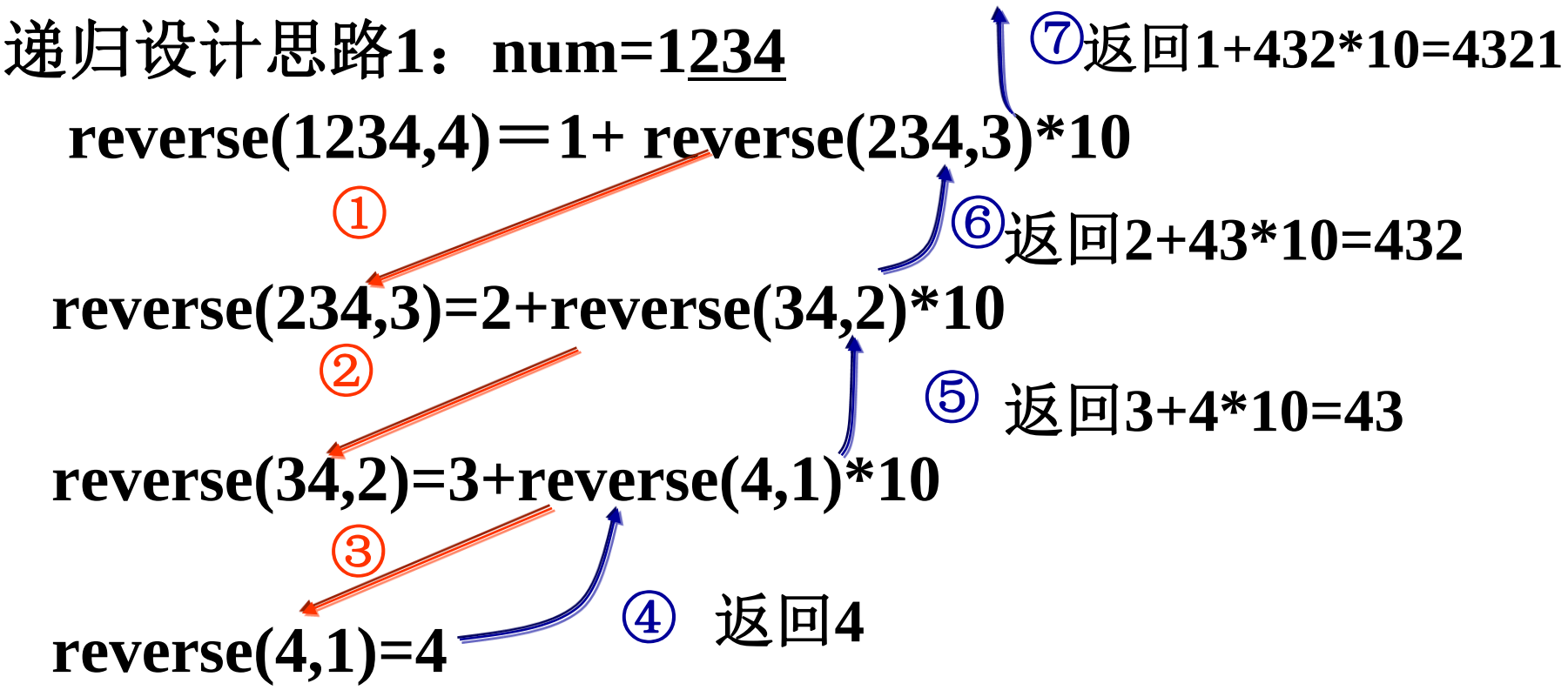
restNum= num % power(10, len-1);

len=num的长度;

递归函数参数设计考虑：len作为递归函数的参数传入

递归程序设计举例

递归设计思路1： num=1234



递归程序设计举例

```
int reverse(int num,int len)
{
    int restNum, highBit;
    if (len == 1)          //如果num是个位数则递归结束
        return num;
    else{
        highBit = num / power(10, len-1);//得到最高位
        restNum = num % power(10, len-1); //得到剩余位
        //递归调用，并形成逆置数
        return highBit + reverse(restNum, len-1)*10;
     }
}
```

num=1234

自定义power函数：int power(int x,int y)

递归程序设计举例

练习3.求任意正整数的逆置数。　　num=1234

递归思路2：将除最低位之外的数先逆置。

例如：reverse(1234)=4*1000+reverse(123)

递归定义1：

reverse (num)=

{ num　　if num长度为1

lowBit*power(10,len-1)+reverse(restNum)　　if num长度大于1

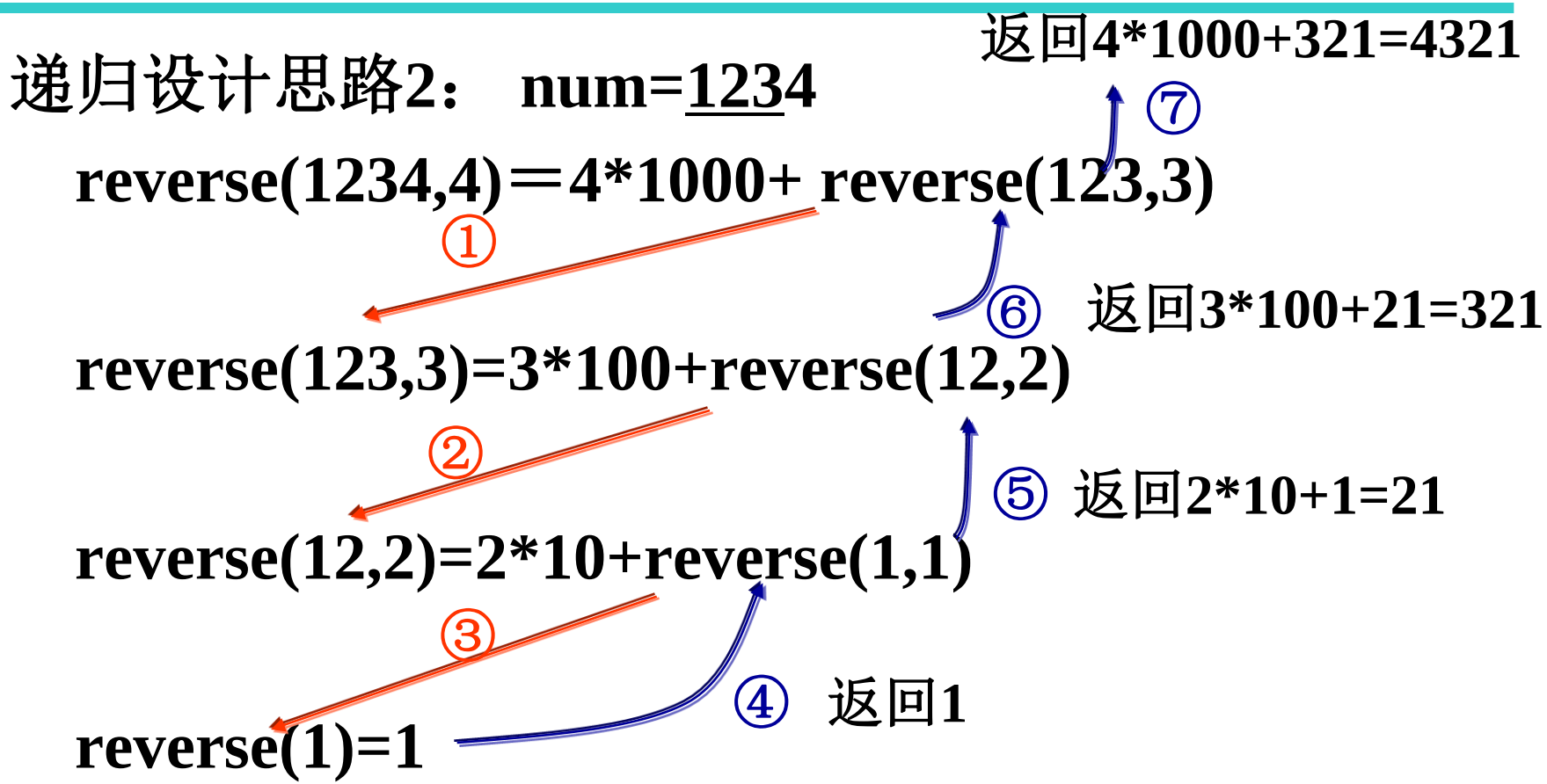
其中：　lowBit=num%10;

restNum=num/10;

len=num的长度;

递归程序设计举例

递归设计思路2： num=1234



递归程序设计举例

num=1234

```
int reverse(int num,int len)
{
    int restNum;
    int lowBit;
    if (len==1)  //余数为0，作为递归的结束条件
      return num;
    else{
      lowBit=num%10;//保留最低位
      restNum=num/10;//得到除去最低位后的余数
      //递归调用，并形成逆置数
      return lowBit*power(10,len-1) + reverse(restNum,len-1);
    }
}
```

递归程序设计举例

练习4. 输入n个整数，求最大数。

递归设计思路1： 15 30 34 10 89

设函数findMax(n)为读取n个数，求最大值

函数Maximum(x,y)为求两个数x和y的最大值

则findMax(n)递归定义1：

- findMax(1)=N_1 当 n值为1
- findMax(n)=Maximum(findMax(n-1),N_n) 当n>1

求前n(n>1)个数的最大值，分解为3步：

第1步：**递推到底，反复**调用n-1；

第2步：读取第n个数num；返回上一层，带回最大值max

第3步：读取 当前数num,返回上一层，带回num和max之间的最大值

计算过程是在回归阶段完成

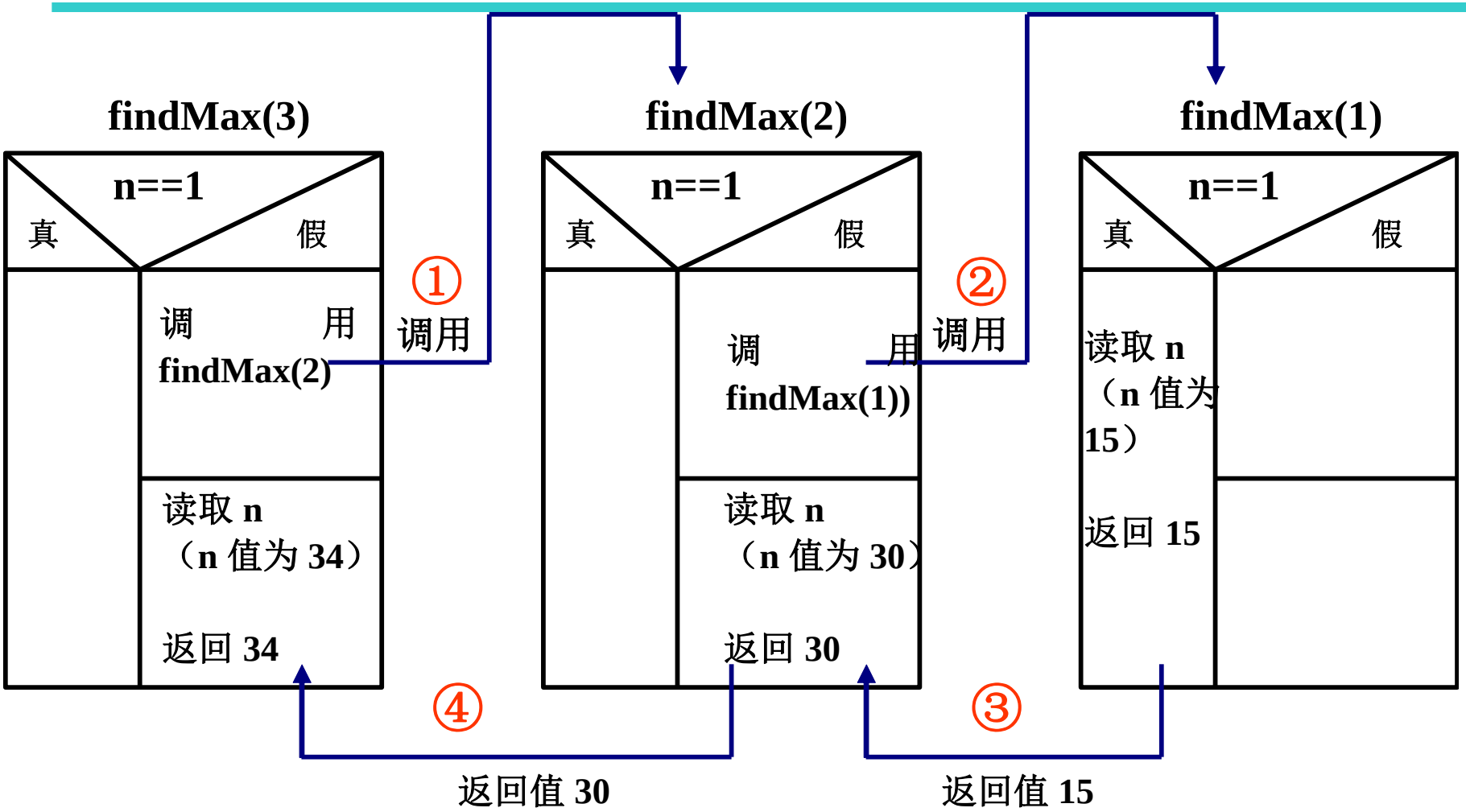
递归程序设计举例

15 30 34 10 89

```
//读取n个数，求最大值
int findMax(int n)
{
    int max;//记录前n-1个数中的最大值
    int num;//读取的第n个值
    if(n==1){//求前1个数中的最大值
        scanf("%d",&num);
        return num;
    }
    else{
       max=findMax(n-1);//第1步：读取前n-1个数，求出最大值max ;
       scanf("%d",&num);//第2步：读取第n个数num;
       return num>max?num:max; /*第3步：计算并返回num和
                                 max之间的最大值*/
  }
}
```

15 30 34 10

```
int findMax(int n)
{
    int max,num;
    if(n==1){//求前1个数中的最大值
        scanf("%d",&num);
        return num;
    }
    else{
        max=findMax(n-1);
        scanf("%d",&num);
        return num>max?num:max;
  }
}
```


findMax(3)
n==1
真
假
调 用 findMax(2)
读取 n
（n 值为 34）
返回 34
①
调用
findMax(2)
n==1
真
假
调 用 findMax(1))
读取 n
（n 值为 30）
返回 30
②
调用
findMax(1)
n==1
真
假
读取 n
（n 值为 15）
返回 15
③
④
返回值 30
返回值 15

练习4. 输入n个整数，求最大数。

递归设计思路2：15 30 34 10 89

设函数findMax(n)为读取n个数，求最大值

函数Maximum(x,y)为求两个数x和y的最大值

则findMax(n)递归定义1：

- findMax(1)=N_1 当 n值为1
- findMax(n)=Maximum(N_1,findMax(n-1)) 当n>1

求前n(n>1)个数的最大值，分解为3步：

第1步：读入第1个数num

第2步：调用递归函数，读取后续n-1个数，求最大数max

第3步：返回num和max中的最大值

```
int findMax (int n) //读取n个数，求最大值
{
    int num, max;
    if (n==1){
        scanf("%d",&num);  /*第1步：读入第一个数num */
        return num;  /*若是最后一个数，则将其本身作为最大值并返回*/
    }
    else{
        scanf("%d",&num);  /*第1步：读入第一个数num */
        max=findMax (n-1);  /*第2步：调用递归函数读取并求出后续n-1个
                                数的最大值max*/
        return num>max?num:max; /*第3步：返回num和max中的最大值*/
    }
}
```

15 30 34 10 89

15 30 34 10


```
int findMax (int n)
{
    int num, max;
    if (n==1){
        scanf("%d",&num);
        return num;
    }
    else{
        scanf("%d",&num);
        max=findMax (n-1);
        return num>max?num:max;
    }
}
```

递归程序设计举例

练习5. 输入任意个整数，以-1结束，求最大数。

递归定义：

findMax(n)递归定义：

- **findMax(1)=N_1　　if　　n==1**
- **findMax(2)=Maximum(N_1 ,findMax(1))**
- **….**
- **findMax(n)=Maximum(N_1,findMax(n-1))**

问题findMax[10,20,-15,-1]可简化成：

Maximum(10, findMax[20,-15,-1])

递归程序设计举例

递归设计思路：

第一步：读入一个数num；

第二步：调用递归函数求出剩下要读入的数的最大值max;

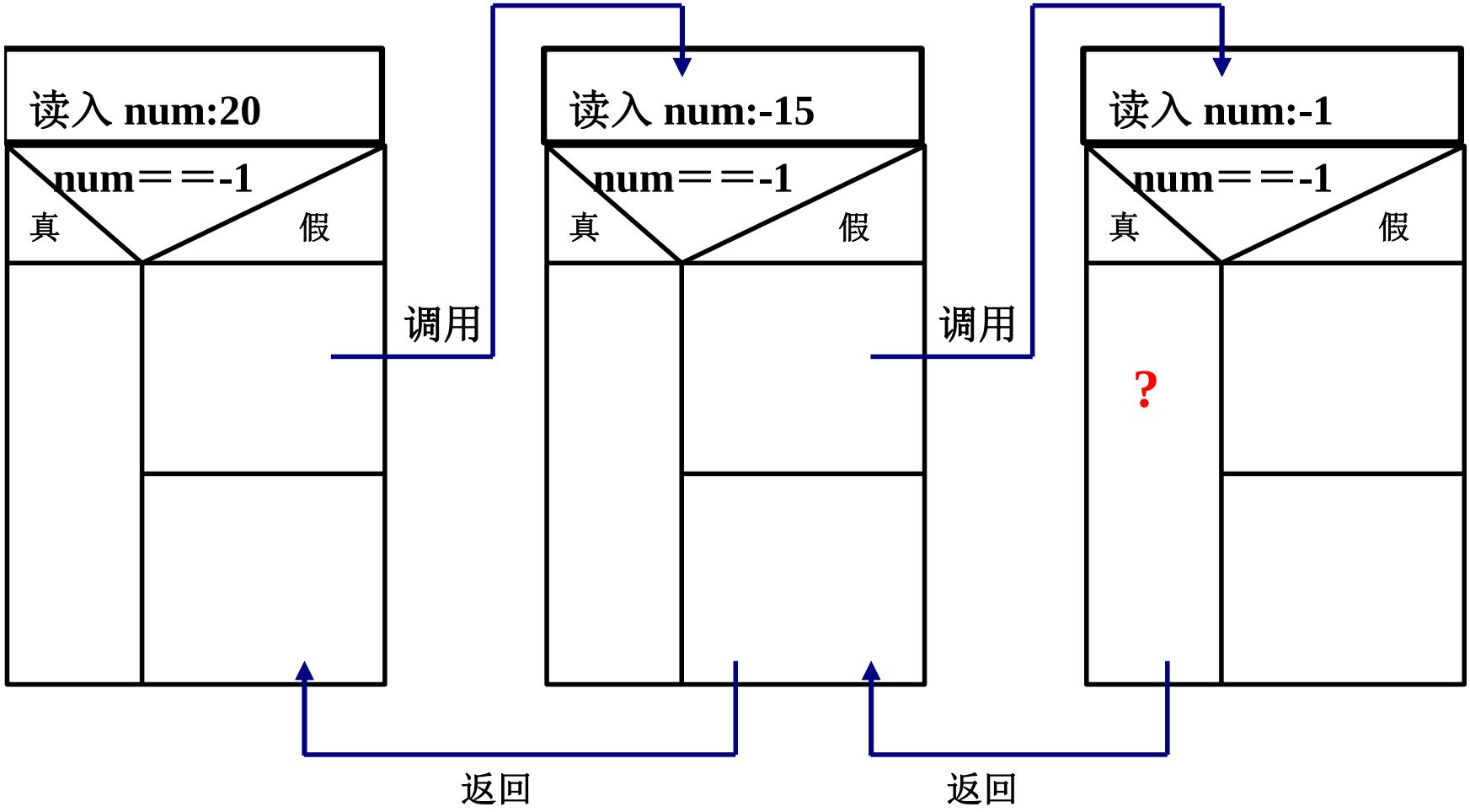
第三步：求出num和max中的最大值并返回（此值即为输入的所有数的最大值）；

递归程序设计举例

```
int findMax()
{
   int num,max;
   scanf("%d",&num); //读入一个数num
   if(num = = -1)
      return ?   ; //此处怎么写?
   else{
       max = findMax();//求出剩下要读入的数的最大值max
       return(num>max?num:max);//返回num和max中的最大值
}
```

findMax()

[20,-15,-1]



[20,-15,-1]

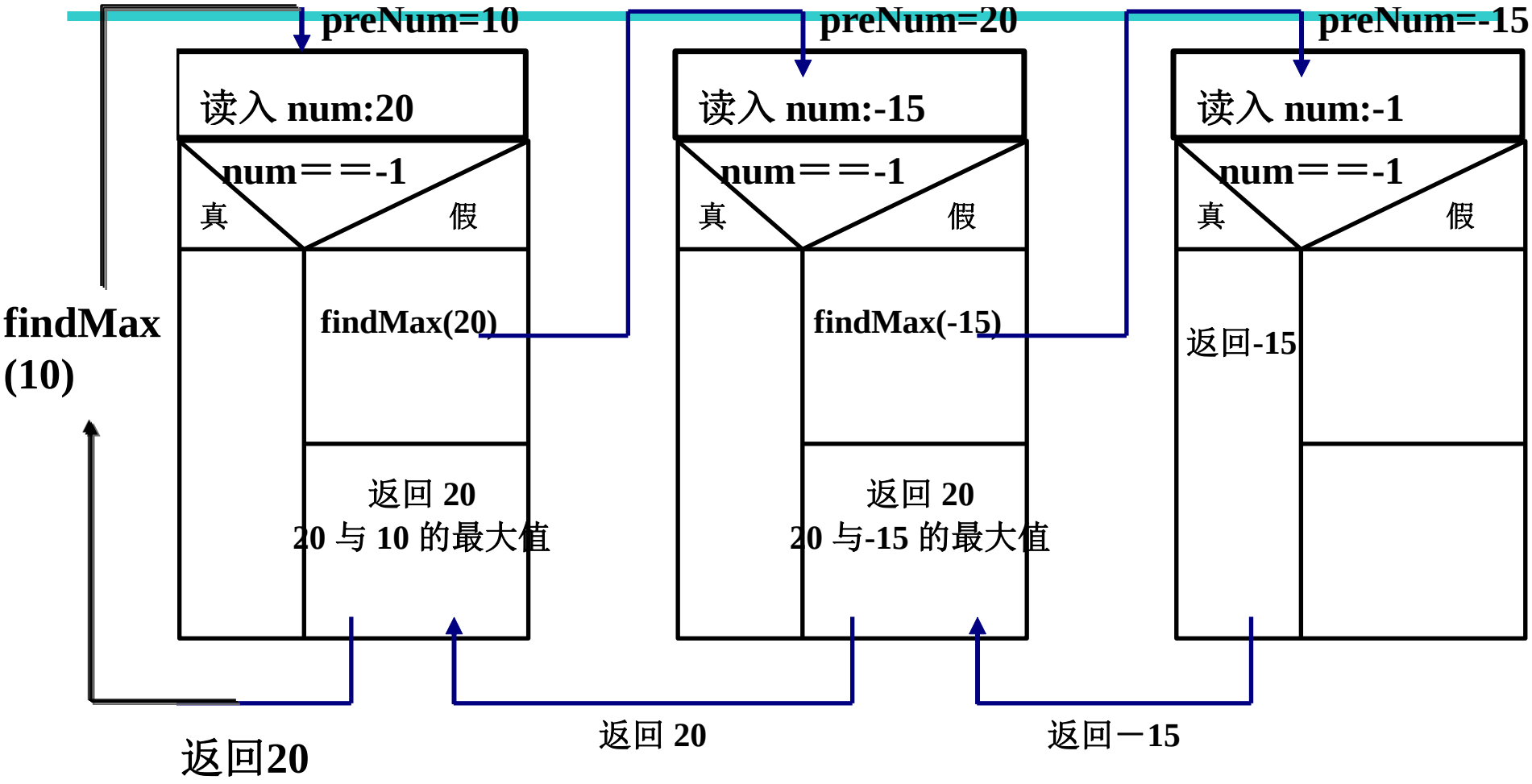
解决方法：将本次递归调用读入的数作为参数传到下一次递归调用中

函数接口定义：**int findMax(int preNum)**

函数功能：求**preNum**和后续读入的任意个数据的最大值

int findMax(int preNum)

[10, 20, -15, -1]



递推过程：

1. 在函数中读入任意数 curr_num,
2. 不断地递归调用函数，将本次读入的curr_num作为函数调用的参数；
3. 直到读入-1；即为递归的结束条件，此时函数返回当前的最大值，即为：curr_num

回归过程：

1. 上一个数preNum与返回的最大值进行比较，确定当前的最大值max，并返回到上一层的函数调用。

递归程序设计举例

```
int findMax(int preNum)//参数为上一层读入的数
{
    int max, curr_num;//当前的最大值，当前层读入的任意数
    scanf("%d",& curr_num);
    if (curr_num ==-1)     //如果是结束标志
        return  preNum ;
   else{
      //递归调用，得到后续数列中的最大值
       max=findMax (curr_num);
     //比较确定当前的最大值
       return (preNum > max? preNum :max);
   }
}
```

例3：用函数fib求斐波那契数列的第n项。斐波那契数列为：0、1、1、2、3、…… 。函数fib定义如下：

$$fib(n)=\begin{cases}0 & \text{当}n=1\\ 1 & \text{当}n=2\\ fib(n-2)+fib(n-1) & \text{当}n>=3\end{cases}$$

```
long fib(long n)
{
    if(n==1 || n==2)
        return n-1;
    else
        return fib(n-2)+fib(n-1);
}
```

请分析该问题是否可以用递归算法解决？若可以，请写出该递归算法。

请画出求fib(4)的递归调用过程图示

```
1.  #include<stdio.h>
2.  main()
3.  {
4.      printf("%ld",f(3));
5.      system("pause");
6.      return 0;
7.  }
8.  long fib(long n)
9.  {
10.     if(n==1 || n==2)
11.         return n-1;
12.     else
13.         return fib(n-2)+fib(n-1);
14. }
```

3.递归程序设计

- 每求一项数，需要递归调用2次该函数；计算斐波那契数列第30项的递归调用次数是2的30次方（大约10亿次！）
- 可见递归的思想特别符合人们的思维习惯，便于问题解决和编程实现。但递归的程序设计方法比较占用系统资源，效率也较低。
- 课下请改写fib函数，使用迭代算法实现

  long fib(long n), 计算斐波那契数列第n项。

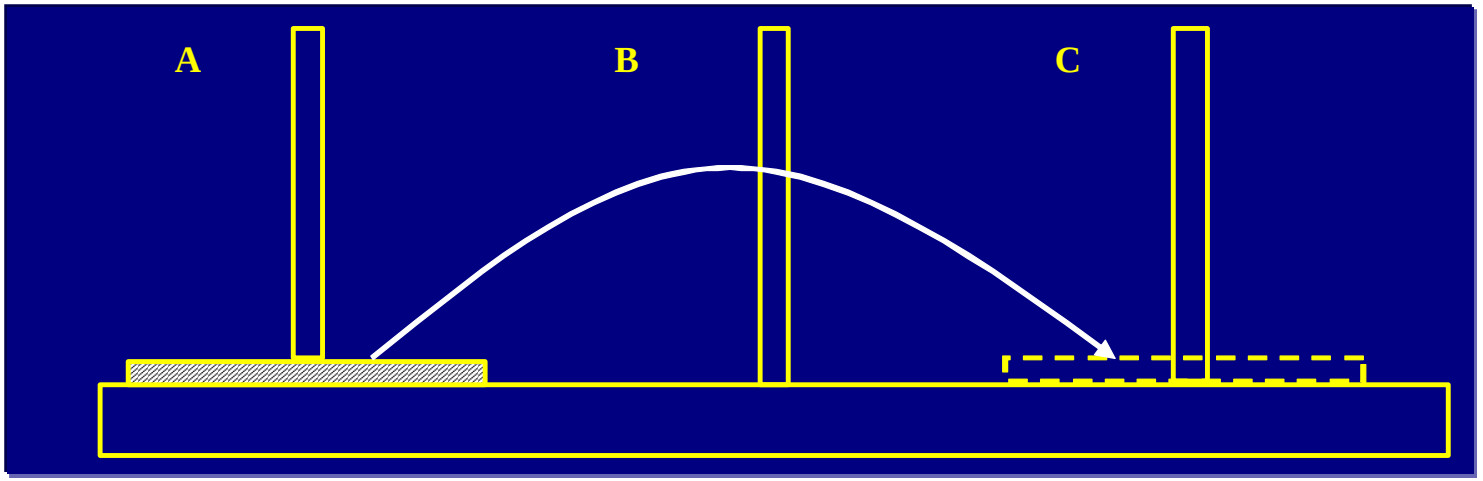
例4、汉诺塔问题

传说印度布拉马圣殿（Temple of Brahma）的教士们有一黄铜烧铸的平台，上立三根金刚石柱子。A柱上堆放了64个金盘子，每个盘子都比其下面的盘子略小一些。当教士们将盘子全部从A柱移到C柱以后，世界就到了末日。当然，这个问题还有一些特定的条件，那就是在柱子之间只能移动一个盘子并且任何时候大盘子都不能放到小盘子上。教士们当然还在忙碌着，因为这需要$2^{64}-1$次移动。如果一次移动需要一秒钟，那么全部操作需要5000亿年以上时间。

怎样编写这种程序？从思路上还是先从最简单的情况分析起，搬一搬看，慢慢理出思路。

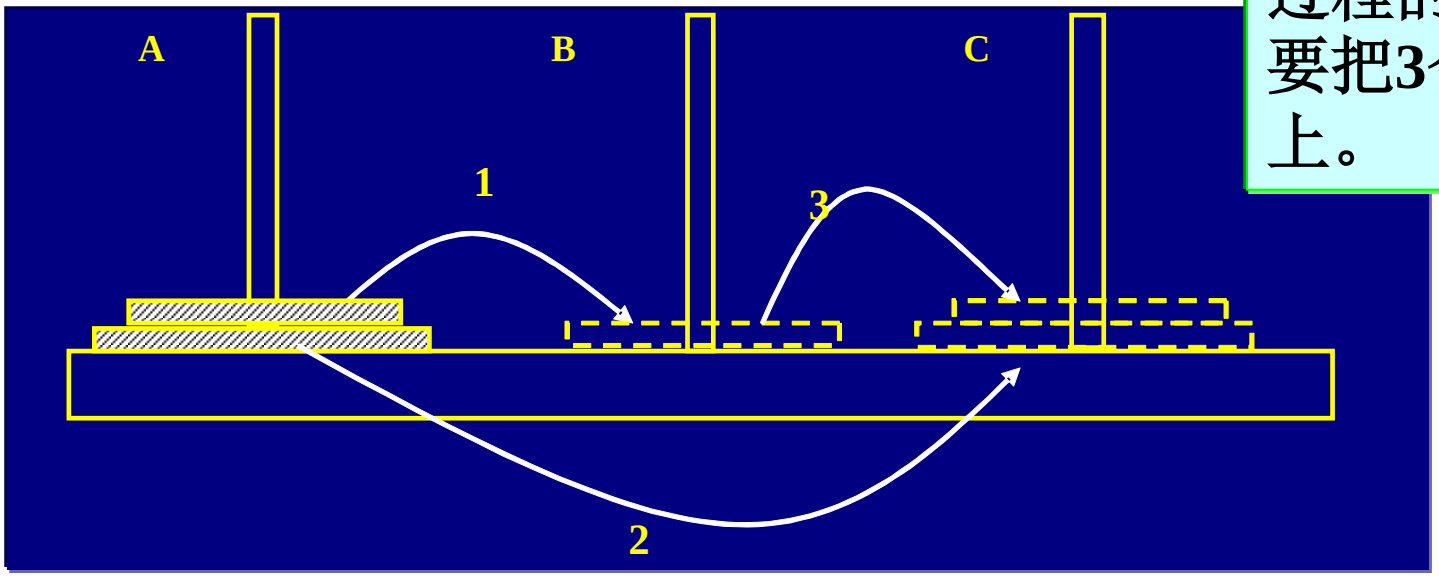
1、在A柱上只有一只盘子，假定盘号为1，这时只需将该盘从A搬至C，一次完成，记为

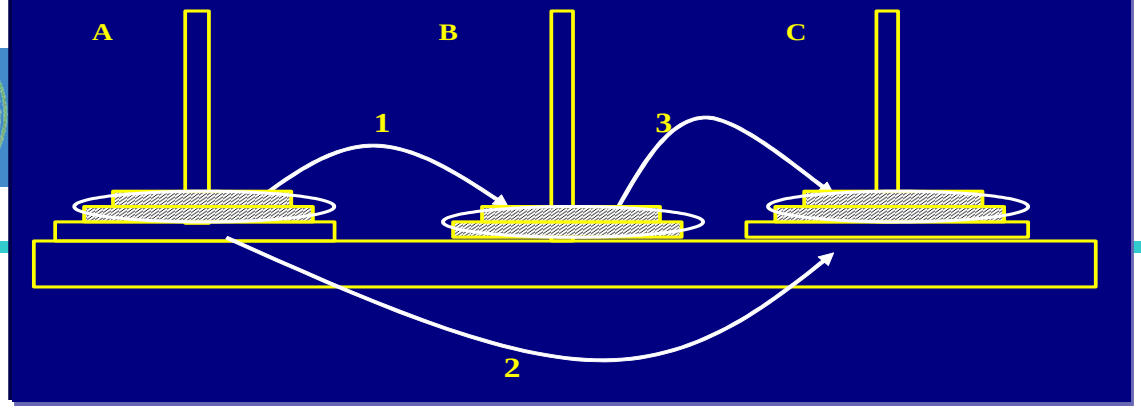
move 1 from A to C



2、在A柱上有二只盘子，1为小盘，2为大盘。
第（1）步将1号盘从A移至B，这是为了让2号盘能移动；
第（2）步将2号盘从A移至C；
第（3）步再将1号盘从B移至C；
这三步记为：move 1 from A to B;
move 2 from A to C;
move 1 from B to C;

可见，移动2个盘子从A柱到C柱需要借助于B柱。因此，在构思搬移过程的参量时，要把3个柱子都用上。





3、在A柱上有3只盘子，从小到大分别为1号，2号，3号

第(1)步将1号盘和2号盘视为一个整体；先将二者作为整体从A移至B，给3号盘创造能够一次移至C的机会。这一步记为 move(2, A, C, B)

意思是将上面的2只盘子作为整体从A借助C移至B。

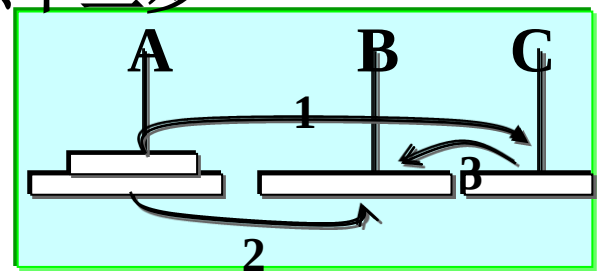
第(2)步将3号盘从A移至C，一次到位。记为

move 3 from A to C

第(3)步处于B上的作为一个整体的2只盘子，再移至C。这一步记为 move(2, B, A, C)

意思是将2只盘子作为整体从B借助A移至C。

4、从题目的约束条件看，大盘上可以随便摞小盘，相反则不允许。在将1号和2号盘当整体从A移至B的过程中**move(2, A, C, B)**实际上是分解为以下三步

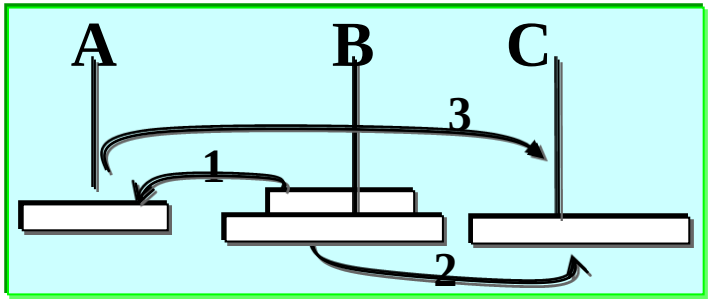


第1步：**move 1 from A to C;**

第2步：**move 2 from A to B;**

第3步：**move 1 from C to B;**

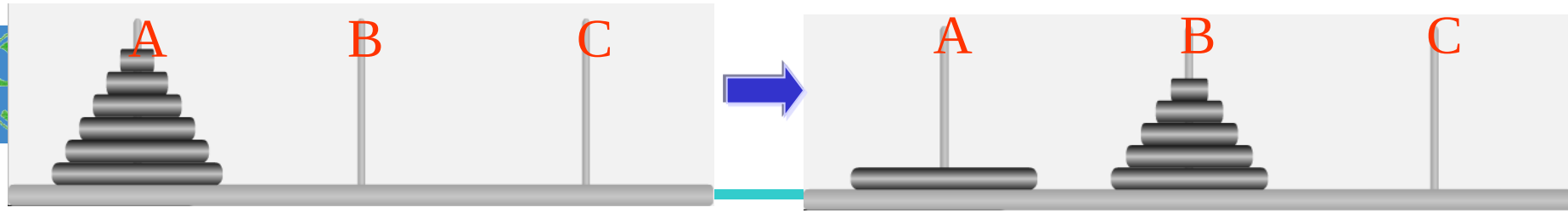
经过以上步骤，将1号和2号盘作为整体从A移至B，为3号盘从A移至C创造了条件。同样，3号盘一旦到了C，就要考虑如何实现将1号和2号盘当整体从B移至C的过程了。实际上**move(2, B, A, C)**也要分解为三步：



第1步：**move 1 from B to A;**

第2步：**move 2 from B to C;**

第3步：**move 1 from A to C;**



同理，将n个圆盘从A柱移到C柱**move(n, A, B, C)**可分解为3步：

第1步.将A柱上面的从上往下数的（n－1）个圆盘移到B柱上，中间通过C柱为辅助。这是一个（n－1）个圆盘的问题：move(n-1,A,C,B)；

第2步. 将A柱上的最后一个圆盘，直接移到C柱上；

第3步.再将B柱上的（n－1）个圆盘移到C柱上，中间以A柱为辅助。这又是一个（n－1）个圆盘的问题:move(n-1,B,A,C)；

这里显然是一种递归定义，将问题分解成若干同样类型的小问题。当解move(n-1, A, C, B)时又可想到，将其分解为3步：

第1步：将上面的n-2只盘子作为一个整体从A经B到C，move(n-2, A, B, C);

第2步：第n-1号盘子从A直接移至B，即n-1:A to B;

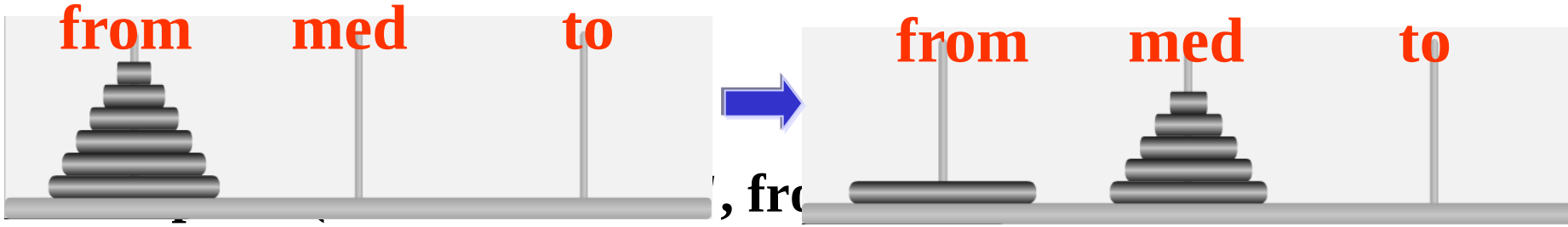
第3步：再将上面的n-2只盘子作为一个整体从C经A移至B，move(n-2, C, A, B);

这样移动n个盘子的问题被简化成了移动n－1个盘子的问题，移动n-1个盘子的问题又被简化成移动n-2个盘子　……这一过程反复进行下去直到只剩下1个盘子时将其移动。移动1个盘子的简单操作就是终止条件。

将n个盘子从from柱移动到to柱，借助于med柱。

1. 将（n－1）个盘子从柱from移到柱med，借助于柱to；
2. 将柱from上的最后一个盘子直接移动到柱to；
3. 将（n－1）个盘子从柱med，移到柱to，借助于柱from；

```
void move(int n, int from, int med, int to)
```



函数接口设计：void move (int n, int from, int med, int to)

功能：将n个盘子从from柱移动到to柱，中间借助于 med柱。

参数：n：要移动的盘子数；from：源柱， med：辅助的柱子，to：目标柱

```
}
```

主函数部分：

```
#include<stdio.h>
void move(int n,int a,int b,int c);
main()
{
   int num;
   printf("the number of plate is:");
   scanf("%d",&num);
   move(num, 1, 2, 3);
   system(“pause”);
   return 0;
 }
```

【程序运行演示】

- 移动3个盘子的递归调用过程见《计算机导论与程序设计基础》教材
- 注意分析
  - 为何调用能正确返回
  - 为何每次调用访问的是正确的函数运行空间

移动三个盘子的运行结果

the number of plate is:3

1-->3

1-->2

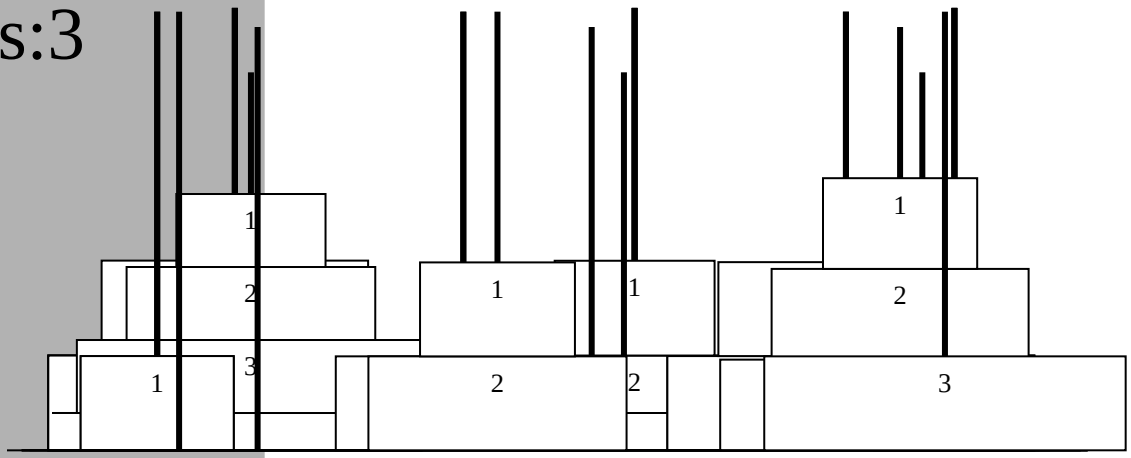
3-->2

1-->3

2-->1

2-->3

1-->3



递归小结

- 递归程序好看、好读，风格优美，但执行效率低；
- 任何能用递归解决的问题都能用迭代（循环）的方法去解决（不过有些问题可能很难写）。
- 终结条件。程序必须要有终结条件，不可能无限递归下去。