二 、算法基本工具和优化技巧

- 利用算法的基本机制——循环和递归设计算法
- 利用数组提高算法质量
- 利用算法的基本操作提高算法效率的技巧

1、循环与递归

1.1 循环设计要点

- 1. 设计中要注意算法的效率
- 2. “自顶向下”的设计方法
- 3. 由具体到抽象设计循环结构

1.1 循环设计要点

- 1．设计中要注意算法的效率
  - 循环体的特点是：“以不变应万变”。
  - 所谓“不变”是指循环体内运算的表现形式是不变的，而每次具体的执行内容却是不尽相同的。

1.1 循环设计要点

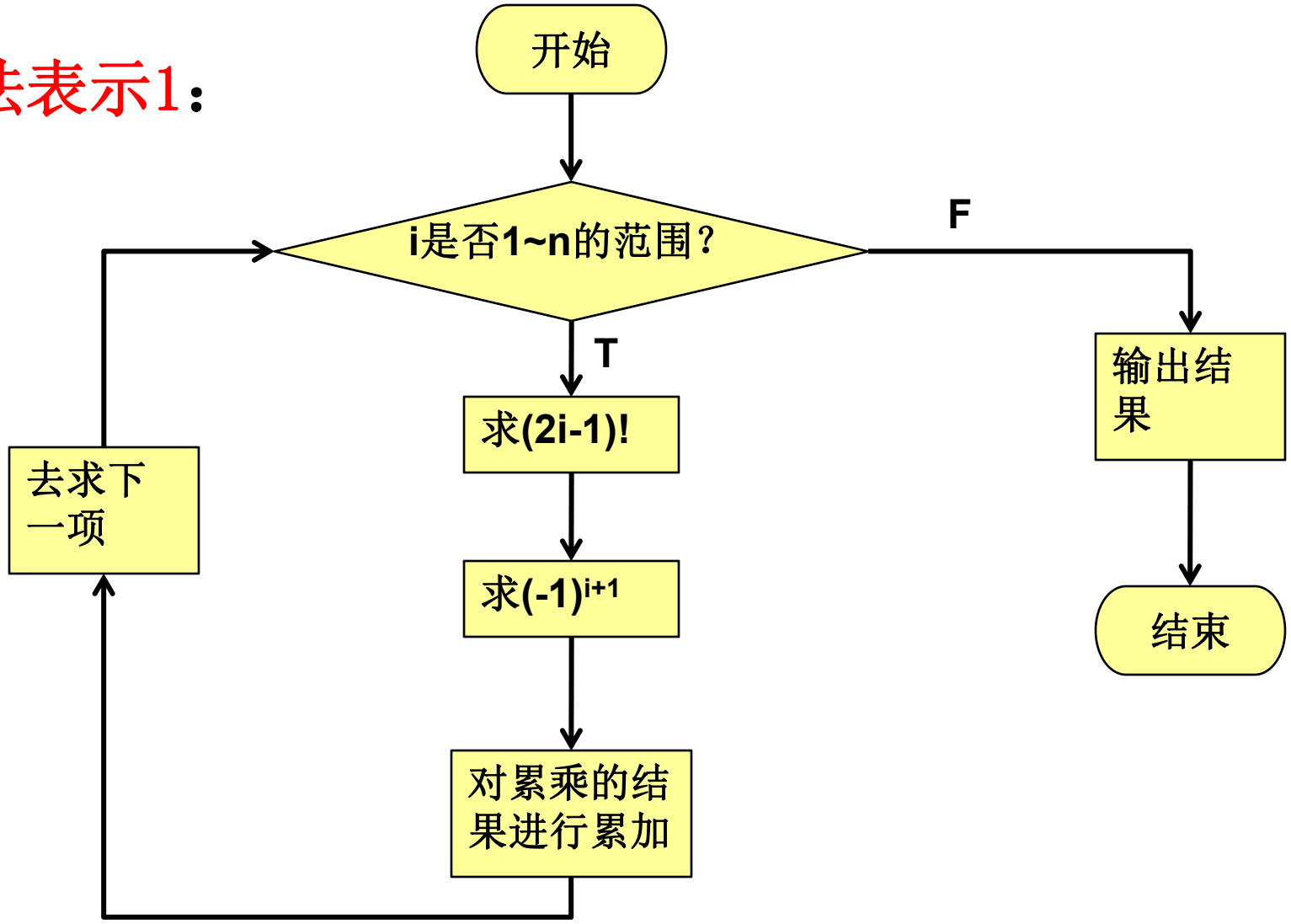
- 例：求1/1!-1/3!+1/5!-1/7!+…+（-1）$^{n+1}$/(2n-1)!
- 问题分析：此问题中既有累加又有累乘，准确地说累加的对象是累乘的结果。
- 数学模型1：$S_n=S_{n-1}$ +（-1）$^{n+1}$/(2n-1)!
- 算法设计1：利用题目中累加项通式，构造出循环体不变式为：

  S=S+（-1）$^{n+1}$/(2n-1)!

  需要用二重循环来完成算法

1.1 循环设计要点

- 算法表示1：



1.1 循环设计要点

- 算法实现1：

```
main( )
{
   int i,n,j,flag;
   float s=1,t;
   scanf("%d",&n);
   for(i=2;i<=n;i++)
   {
        t=1;
        for(j=1;j<=2*i-1;j++)
               t=t*j;     //求(2n-1)!
        flag =1;
        for(j=1;j<=i+1;j++)
               flag=flag*(-1);    //求-1的n+1次方
        s=s+ flag/t;
   }
   printf("SUM=%f\n",s);       //n=5  SUM=0.841471
}
```

1.1 循环设计要点

- 算法分析：
- 以上算法是由二重循环来完成 ，但算法的效率太低为0（n^2）。
- 其原因是，当前一次循环已求出7！，当这次要想求9！时，没必要再从1去累乘到9，只需要充分利用前一次的结果，用7！*8*9即可得到9！。
- 数学模型2：$S_n=S_{n-1}+(-1)^{n+1}/A_n$；
  $A_n=A_{n-1}*((2*n-2)*(2*n-1))$
- 另外运算flag= flag*(-1)效率也是0（n^2） ，这也是没有必要的。

1.1 循环设计要点

- 算法实现2：

```
main( )
{
   int i,n,flag=1;
   float s=1,t=1;
   scanf("%d",&n);
   for(i=2;i<=n;i++)
   {
       flag=-flag;
       t=t*(2*i-2)*(2*i-1);
       s=s+flag/t;
   }
   printf("SUM=%f\n",s);
}
```

1.1 循环设计要点

- 算法分析：
- 按照数学模型2，只需一重循环就能解决问题，算法的时间复杂度为0（n）。

1.1 循环设计要点

- 2. “自顶向下”的设计方法
  - 自顶向下的方法是从全局走向局部、从概略走向详尽的设计方法。
  - 自上而下是系统分解和细化的过程。

1.1 循环设计要点

- 例：编算法找出1000以内所有完数
  - 例如，28的因子为1、2、4、7、14，而28=1+2+4+7+14。因此28是“完数”。编算法找出1000之内的所有完数，并按下面格式输出其因子：“完数28的因子有1，2，4，7，14”
  - 这里不是要质因数，所以找到因数后也无需将其从数据中“除掉”。
  - 每个因数只记一次。（注：本题限定因数不包括这个数本身）

1.1 循环设计要点

- 1）顶层算法
  - for(i=1;i<=1000;i++)　//确定要寻找的完数的范围
- 2）确定i的所有可能因数的范围（除i本身之外）
  - for(j=1;j<i;j++)
- 3）进一步细化——判断i是否“完数”
  - 如果j是i的因数，则累加j，s=s+j;
  - 累加完之后，如果i==s，则i是“完数”；

1.1 循环设计要点

- 4）考虑输出格式——判断i是否“完数”
- 考虑到要按格式输出结果，应该开辟数组存储数据i的所有因数，因此算法细化如下：
  - 如果j是i的因数，则累加j，s=s+j;
  - 记录j到数组
  - 累加完之后，如果i==s，则i是“完数”；

1.1 循环设计要点

- 算法实现：

```
main()
{  int i,k,j,s,a[30];
   for(i=1;i<=1000;i++)
   {    s=0;
        k=0;
        for(j=1;j<i;j++)  //j是所有可能因数范围（除本身之外）
                if(i%j==0)
                {       s=s+j;          a[k]=j;         k++;}
        if(i==s)
        {
                printf("完数%d的因子有：",s);
                for(j=0;j<k;j++)
                        printf(",%d",a[j]);
                printf("\n");
        }
   }
 }
```

1.1 循环设计要点

- 例：求一个矩阵的鞍点(即在行上最小而在列上最大的点)。
- 算法设计：
  - 1）在第一行找最小值，并记录其列号。
  - 2）然后验证其是否为所在列的最大值，如果是，则找到问题的解；否则，则继续在下一行找最小值 ……

1.1 循环设计要点

- 算法实现：

```
main( )
{
   int
   a[4][5]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
   int  i,j,k,minj,t,kz=0;  //minj代表当前行中最小值的列下标
   for(i=0;i<4;i++)
   {
        t=a[i][0];  //每行第一个当擂主
        minj=0;
        for(j=1;j<5;j++)
               if(a[i][j]<t)  //与该行后面每一个比
               {
                      t=a[i][j];  //找到该行最小的记录到t
                      minj=j;  //记录列下标到minj
               }
```

1.1 循环设计要点

- 算法实现：

```
        for(k=0;k<4;k++)
                if(a[k][minj]>a[i][minj])  //该列中如果有比当前行更大的则退出该列的判断
                        break;
        if(k<4)    //该列中出现了比当前行更大的，则去下一行找鞍点
                continue;
        printf("矩阵鞍点为a[%d][%d]=%d\n",i,minj,a[i][minj]);  //如果当前行是该列中最大的，则作为鞍点输出
        return;
    }
    printf("没有鞍点\n");
}
```

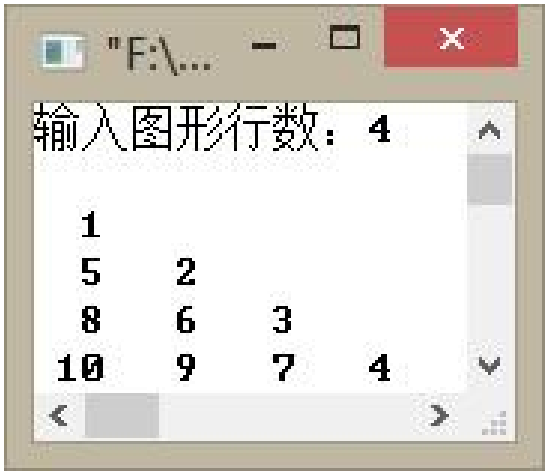
1.1 循环设计要点

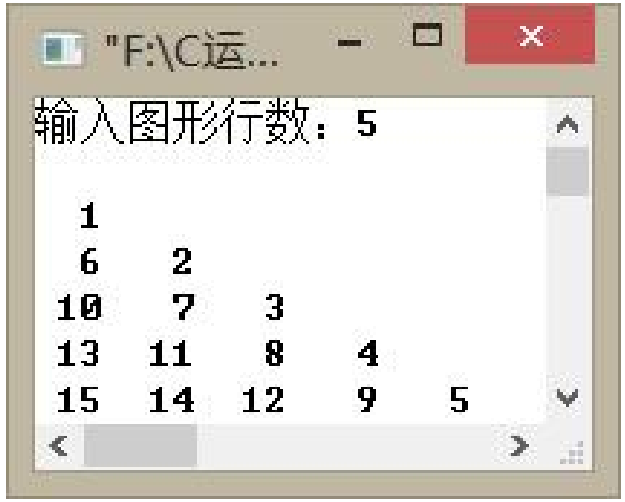
- 3. 由具体到抽象设计循环结构
  - 对于不太熟悉的问题，其数学模型或“机械化操作步骤”不易抽象。下面看一个由具体到抽象设计循环细节的例题。

1.1 循环设计要点

- 例：编写算法，打印具有下面规律的图形。

```
1
5  2
8  6  3
10 9  7  4
```





1.1 循环设计要点

- 算法设计：
  - 1，2，3，4所在位置称为第一层（主对角线），5，6，7所在位置称为第二层，……
  - 第一层有n个元素，第二层有n-1个元素……
  - 以层号作为外层循环，循环变量为i
  - 以层内元素从左上到右下的序号作为内循环，循环变量为j（第一层循环n次，第二层循环n-1次，第三层循环n-2次……）
  - 这样循环的执行过程正好与“摆放”自然数的顺序相同
  - 用具体到抽象设计循环的“归纳法”，找出数组元素的行号、列号与层号i及层内序号j的关系

1.1 循环设计要点

- 算法设计：
  - 每层内元素的列号都与其所在层内的序号j是相同的。
    - 第一层：1（列号0）、2（列号1）、3（列号2）、4（列号3）
    - 第二层：5（列号0）、6（列号1）、7（列号2）
    - 第三层：……
  - 元素的行与其所在的层号及在层内的序号均有关系：
    - 第一层：1（行号0）、2（行号1）、3（行号2）、4（行号3）
    - 第二层：5（行号1）、6（行号2）、7（行号3）
    - 第三层：8（行号2）、9（行号3）
    - 第四层：……

1.1 循环设计要点

- 算法实现：

```
main( )
{
   int i,j,a[100][100],n,k;
   printf("输入图形行数：");
   scanf("%d",&n);
   k=1;
   for(i=0;i<n;i++)
        for(j=0;j<n-i;j++)
        {       a[i+j][j]=k;   k++;    }
        for(i=0;i<n;i++)
        {
               printf("\n");
               for( j=0;j<=i;j++)
                      printf("%3d ",a[i][j]);
        }
}
```

1.2 递归设计要点

- 递归算法是一个函数除了可调用其它函数外，还可以直接或间接地调用自身的算法。
- 递归是一种比迭代循环更强、更好用的循环结构。
- 只需要找出递归关系和最小问题的解。
- 递归方法只需少量的步骤就可描述出解题过程所需要的多次重复计算，大大地减少了算法的代码量。

1.2 递归设计要点

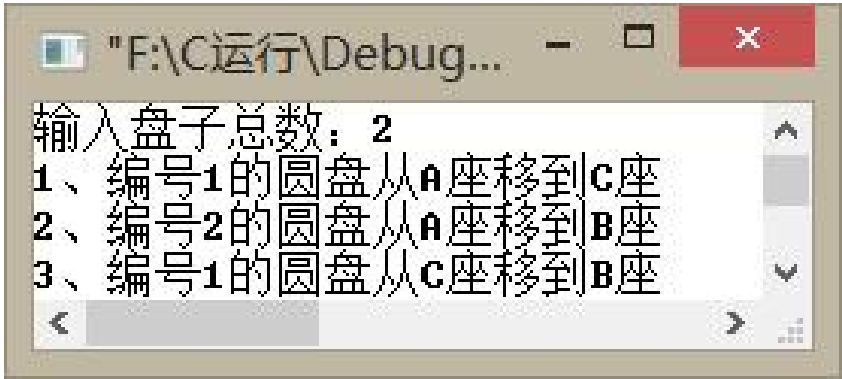
- 递归的关键在于找出递归方程式和递归终止条件，三个步骤：
  - 1、分析问题、寻找递归：找出大规模问题与小规模问题的关系，这样通过递归使问题的规模逐渐变小。
  - 2、设置边界、控制递归：找出停止条件，即算法可解的最小规模问题。
  - 3、设计函数、确定参数：和其它算法模块一样设计函数体中的操作及相关参数。
- 经典的递归例题：
  - 汉诺塔问题

1.2 递归设计要点

- 例：汉诺塔问题描述：
- 古代有一个梵塔，塔内有3个基座A、B、C，开始时A基座上有64个盘子，盘子大小不等，大的在下，小的在上。有一个老和尚想把这64个盘子从A座移到B座，但每次只允许移动一个盘子，且在移动过程中在3个基座上的盘子都始终保持大盘在下，小盘在上。在移动过程中可以利用C基座做辅助。请编程打印出移动过程 。

1.2 递归设计要点

- 算法设计：
- 用归纳法解此题，约定盘子自上而下的编号为1，2，3，……，n。
- 2阶汉诺塔问题的解，移动过程如下：
  - 初始状态 A（1,2） B（） C（）
  - 第一步后 A（2） B（） C（1）
  - 第二步后 A（） B（2） C（1）
  - 第三步后 A（） B（1,2） C（）

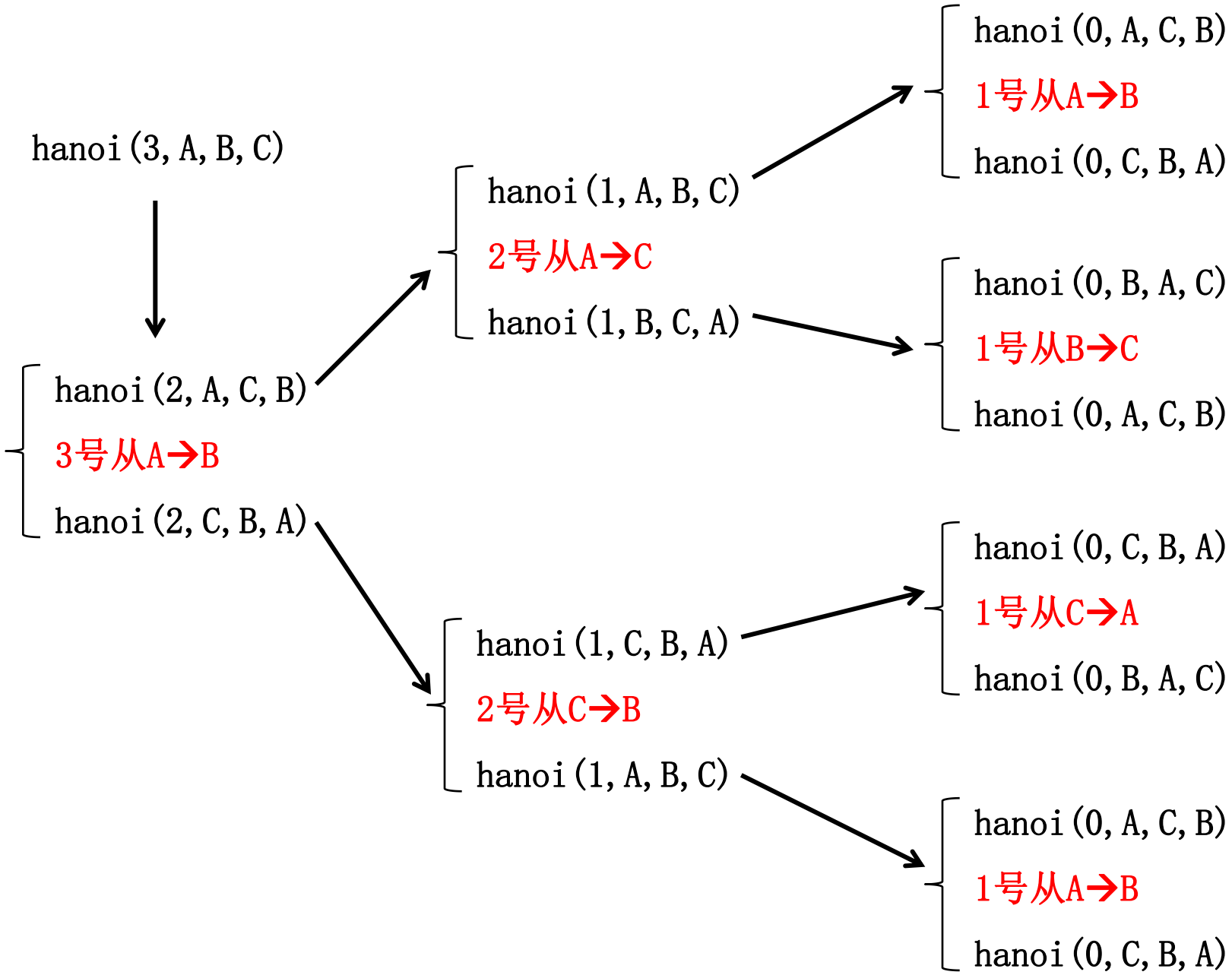


1.2 递归设计要点

- 找出大规模问题与小规模问题的关系，从而实现递归
- 把n个盘子抽象地看作“两个盘子”，上面“一个”由1——n-1号组成，下面“一个”就是n号盘子。移动过程如下：
  - 第一步：先把上面“一个”盘子以A基座为起点借助B基座移到C基座。
  - 第二步：把下面“一个”盘子从A基座移到B基座。
  - 第三步：再把C基座上的n-1盘子借助A基座移到B基座。

1.2 递归设计要点

- 把n阶的汉诺塔问题的函数记作hanoi(n,a,b,c)
  - a代表每一次移动的起始基座
  - b代表每一次移动的终点基座
  - c代表每一次移动的辅助基座
- 则汉诺塔问题等价于以下三步：
  - 第一步，hanoi(n-1,A,C,B)；
  - 第二步，把下面“一个”盘子从A基座移到B基座；
  - 第三步，hanoi(n-1,C,B,A)。


hanoi (3, A, B, C)
hanoi (2, A, C, B)
3号从A→B
hanoi (2, C, B, A)
hanoi (1, A, B, C)
2号从A→C
hanoi (1, B, C, A)
hanoi (1, C, B, A)
2号从C→B
hanoi (1, A, B, C)
hanoi (0, A, C, B)
1号从A→B
hanoi (0, C, B, A)
hanoi (0, B, A, C)
1号从B→C
hanoi (0, A, C, B)
hanoi (0, C, B, A)
1号从C→A
hanoi (0, B, A, C)
hanoi (0, A, C, B)
1号从A→B
hanoi (0, C, B, A)

1.2 递归设计要点

- 算法实现：

```
#include<windows.h>
int i=0;
hanoi (int n,char a,char b,char c)
{
   if(n>0)
   {    hanoi(n-1,a,c,b);
        printf("%d、编号%d的圆盘从%c座移到%c座\n",++i,n,a,b);
        Sleep(1000);
        hanoi(n-1,c,b,a);
   }
}
main()
{
   char a='A',b='B',c='C';
   int n;
   printf("输入盘子总数：");
   scanf("%d",&n);
   hanoi(n,a,b,c);
}
```

1.3 递归与循环的比较

- 例：任给十进制的正整数，请从低位到高位逐位输出各位数字。
- 循环算法设计：
  - 1）求个位数字：n%10
  - 2）为了保证循环体为“不变式”，求十位数字仍然为n%10，这就要通过n=n/10，将n的十位数变成个位数。

1.3 递归与循环的比较

- 算法实现：

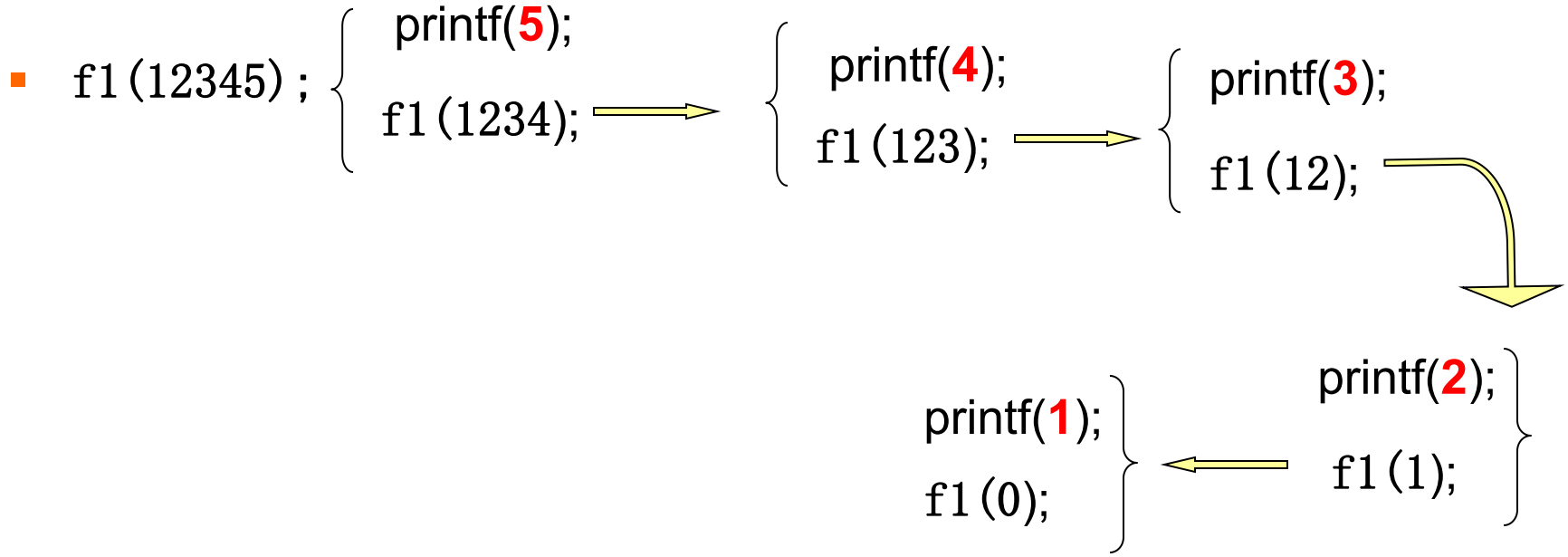
```
f1(int n)
{
   while(n>0)
   {
        printf("%d",n%10);
        n=n/10;
   }
}
main()
{
   int n;
   printf("输入一个正整数：");
   scanf("%d",&n);
   f1(n);
}
```

1.3 递归与循环的比较

- 递归算法设计：
  - 1）求个位数字n%10，下一步则是递归地求n/10的个位数字。
  - 2）当n=0时，则所有位上的数都已经求完。

1.3 递归与循环的比较

- 设n=12345



1.3 递归与循环的比较

- 算法实现：

```
f1(int n)
{
   if(n>0)
   {
        printf("%d",n%10);
        f1(n/10);
   }
}
main()
{
   int n;
   printf("输入一个正整数：");
   scanf("%d",&n);
   f1(n);
}
```

1.3 递归与循环的比较

- 例：任给十进制的正整数，请从高位到低位逐位输出各位数字。
- 循环算法设计：
  - 算法还是“从低位到高位”逐位求出各位数字比较方便。但是就不能边计算边输出，需要用数组保存计算结果，最后倒着输出。

1.3 递归与循环的比较

- 算法实现：

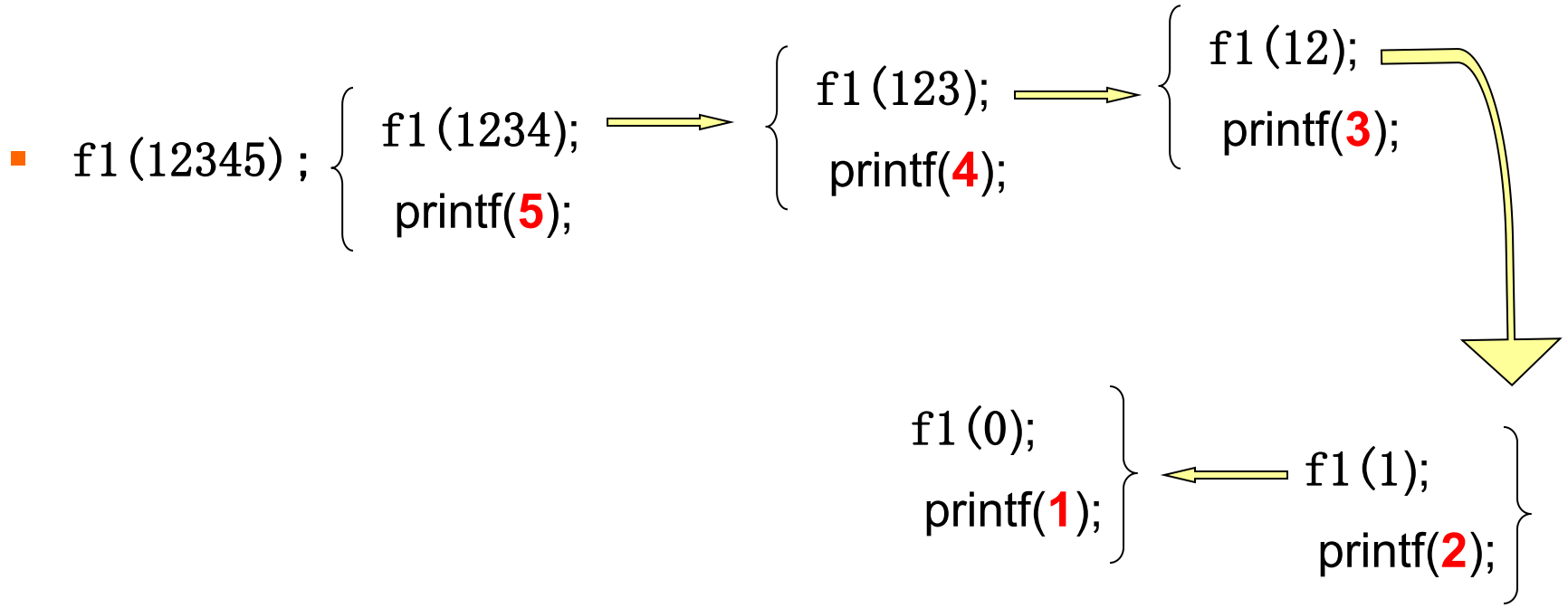
```
f1(int n)
{
   int a[20],i=0,j;
   while(n>0)
   {    a[i]=n%10;
        n=n/10;
        i++;
   }
   for(j=i-1;j>=0;j--)
        printf("%d",a[j]);
}
main()
{  int n;
   printf("输入一个正整数：");
   scanf("%d",&n);
   f1(n);
}
```

1.3 递归与循环的比较

- 递归算法设计：
  - 1）先递归地求n的“非个位数字”（n/10 ），然后再求“非个位数字” n的个位数字（ n%10 ）并输出，输出操作是在回溯时完成的。
  - 2）当n=0时，则所有位上的数都已经求完。

1.3 递归与循环的比较

- 设n=12345



1.3 递归与循环的比较

- 算法实现：

```
f1(int n)
{
   if(n>0)
   {
        f1(n/10);
        printf("%d",n%10);
   }
}
main()
{
   int n;
   printf("输入一个正整数：");
   scanf("%d",&n);
   f1(n);
}
```

1.3 递归与循环的比较

- 由于递归算法的实现包括递归和回溯两步，当问题需要“后进先出”的操作时，用递归算法则更有效。如数据结构课程中树的各种遍历、图的深度优先等算法都是如此。
- 无论把递归作为一种算法的策略，还是一种实现机制，对我们设计算法都有很好的帮助。

1.3 递归与循环的比较

	递归	非递归
程序可读性	易	难
代码量大小	小	大
时间	长	短
占用空间	大	小
试用范围	广	窄
设计难度	易	难