



浙江财经大学

Zhejiang University Of Finance & Economics



递归初步

信智学院 陈琰宏



教学内容



01

理解递归

02

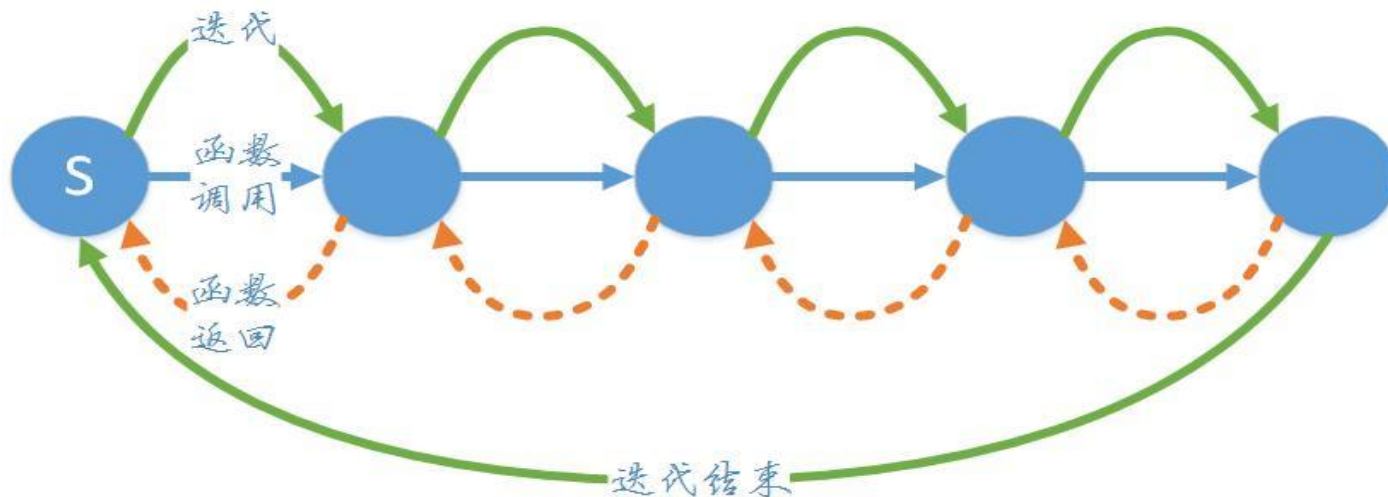
熟练应用递归解决一些实际问题

03

体验递归的审题分析和细节测试

1 递归概念

- 递归是一种重要的算法思想。
- 递归特点是：函数或过程调用它自己本身。其中直接调用自己称为直接递归，而将A调用B，B以调用A的递归叫做间接递归。
- 递归既可以实现**递推过程**，也可以实现求解诸多问题的通用思路—**搜索**。





1.1什么是递归

在数学上，求n的阶乘，有两种表示方法：

① $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$

② $n! = n \times (n-1)!$

这两种表示方法实际上对应到两种不同的算法思想。

第①种表示方法中，求n! 要反复把1、2、3、 $\cdots$ 、(n-2)、(n-1)、n累乘起来，是循环的思想，要用**循环结构**来实现。

第②种表示方法，求n! 时需要用到(n-1)! 。如果有一个函数 **Factorial(int n)** 能实现求n的阶乘，则该函数在求n! 时要使用到表达式： **$n \times \text{Factorial}(n-1)$** ，Factorial(n-1)表示调用Factorial()函数去求(n-1)! 。

例1.1 递归求阶乘

第②种表示方法，求 $n!$ 时需要用到 $(n-1)!$ 。如果有一个函数 **Factorial(int n)** 能实现求 n 的阶乘，则该函数在求 $n!$ 时要使用到表达式： **$n * \text{Factorial}(n-1)$** ， $\text{Factorial}(n-1)$ 表示调用 $\text{Factorial}()$ 函数去求 $(n-1)!$ 。具体代码如下：

```
int Factorial( int n )
{
    if( n<0 ) return -1;
    else if( n==0 || n==1 ) return 1;
    else return n*Factorial(n-1);
}
int main( )
{
    int n; scanf( "%d", &n );
    printf( "%d!=%d\n", n, Factorial(n) );
    return 0;
}
```

$\text{Factorial}()$ 调用了 $\text{Factorial}()$ 函数，这种函数称为**递归函数**
//递归调用Factorial函数

该程序的运行示例如下：

4✓
4!=24

1.1什么是递归

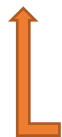
```
16 int main()
```

```
17 {
```

```
18     cout<<fac3(3);
```

```
19     return 0;
```

```
20 }
```



```
12 int fac3(int n){
```

```
13     return n*fac2(n-1);
```

```
14 }
```



```
8 int fac2(int n){
```

```
9     return n*fac1(n-1);
```

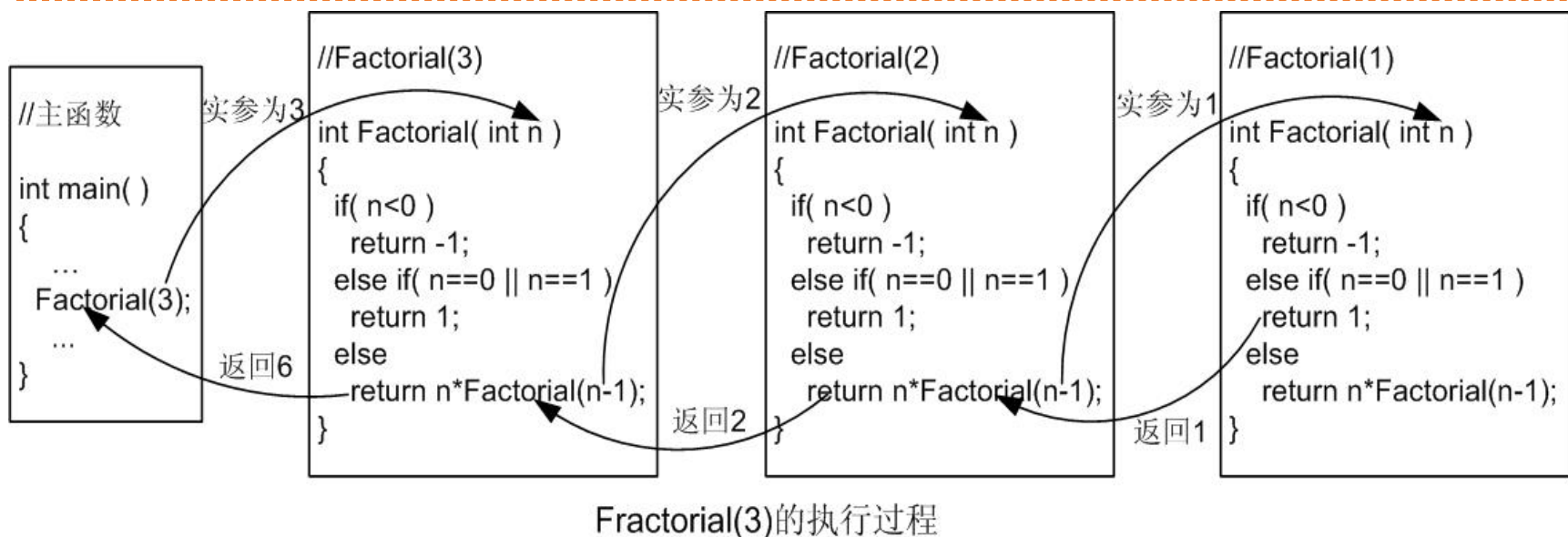
```
10 }
```



```
4 int fac1(int n){
```

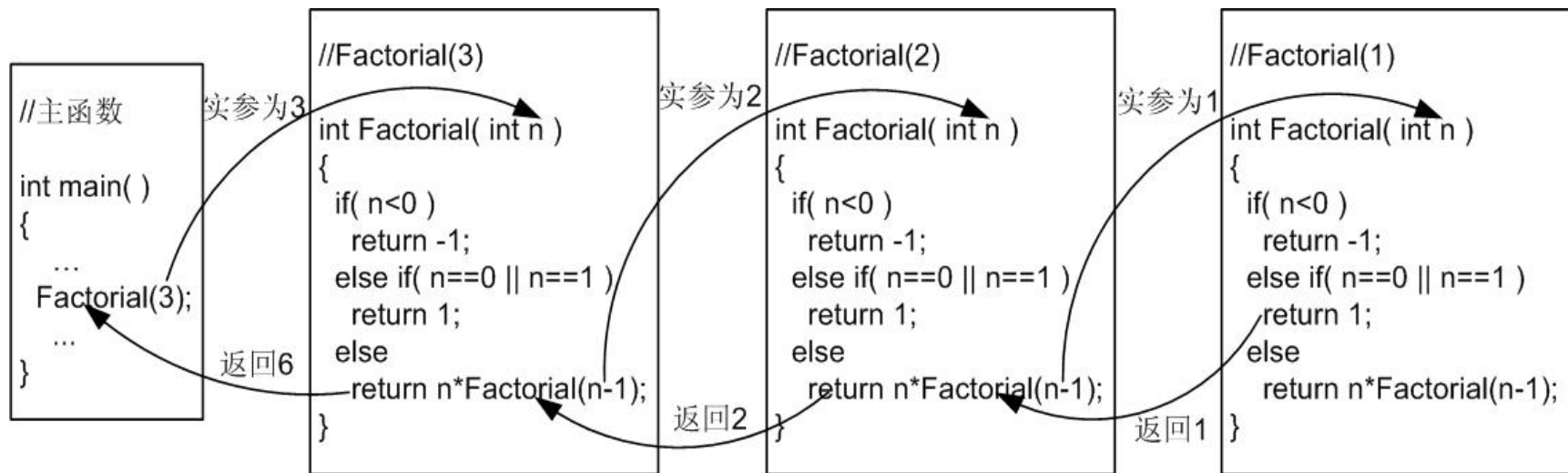
```
5     return 1;
```

```
6 }
```



假设要求3!，其完整的执行过程如图所示，具体过程为：

- ①执行**main**函数的开头部分；
- ②当执行到Factorial函数调用 “**Factorial(3)**”时，流程转而去执行**Factorial(3)**函数，并将实参3传递给形参n；
- ③执行**Factorial(3)**函数的开头部分；
- ④当执行到递归调用Factorial(n-1)函数时，此时n-1=2，所以要转而去执行**Factorial(2)**函数；
- ⑤执行**Factorial(2)**函数的开头部分；
- ⑥当执行到递归调用Factorial(n-1)函数时，此时n-1=1，所以要转而去执行**Factorial(1)**函数；



Factorial(3)的执行过程

- ⑦执行**Factorial(1)**函数，此时因为n的值为1，所以返回1，而不是再递归调用下去，即，Factorial(1)函数执行完毕，返回到上一层，即返回**Factorial(2)**函数中；
- ⑧执行完**Factorial(2)**函数的剩余语句，返回到**Factorial(3)**函数中；
- ⑨执行完**Factorial(3)**函数的剩余语句，返回到**main**函数中；
- ⑩继续执行**main**函数的剩余部分直到整个程序执行完毕。

[2470] 阶乘

输入n，请用递归的方式求n的阶乘。

```
1  #include <iostream>
2  using namespace std;
3  int f(int x)
4  {
5      if(x==1 || x==0) return 1;
6      return x*f(x-1);
7  }
8
9  int main()
10 {
11     int n;
12     cin>>n;
13     cout<<f(n)<<endl;
14     return 0;
15 }
```



[2739]猴子吃桃问题

例1.2。猴子第1天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个。第2天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半另加一个。到第10天早上想再吃时，就只剩下一个桃子了。求第1天共摘了多少个桃子。

分析：

假设 A_i 为第 i 天吃完后剩下的桃子的个数，**A1表示第一天共摘下的桃子**，本题要求的是 A_1 。有以下递推式子：

$A_1 = 2 \times (A_2 + 1)$ A_1 ：第1天吃完后剩下的桃子数

$A_2 = 2 \times (A_3 + 1)$ A_2 ：第2天吃完后剩下的桃子数

.....

$A_9 = 2 \times (A_{10} + 1)$ A_9 ：第9天吃完后剩下的桃子数

$A_{10} = 1$

以上递推过程可分别用**循环结构**和**递归函数**实现。

用递归思想实现：

前面所述的递推关系也可以采用下面的方式描述。假设第n天吃完后剩下的桃子数为 $A(n)$ ，第n+1天吃完后剩下的桃子数为 $A(n+1)$ ，则存在的递推关系： $A(n) = (A(n+1) + 1) * 2$ 。这种递推关系可以用递归函数实现，代码如下：

```
int A(int n)
{
    if(n>=10) return 1;
    else return(2*(A(n+1)+1));
}
int main( )
{
    printf( "total=%d\n", A(1) );
    return 0;
}
```

该程序的输出结果：

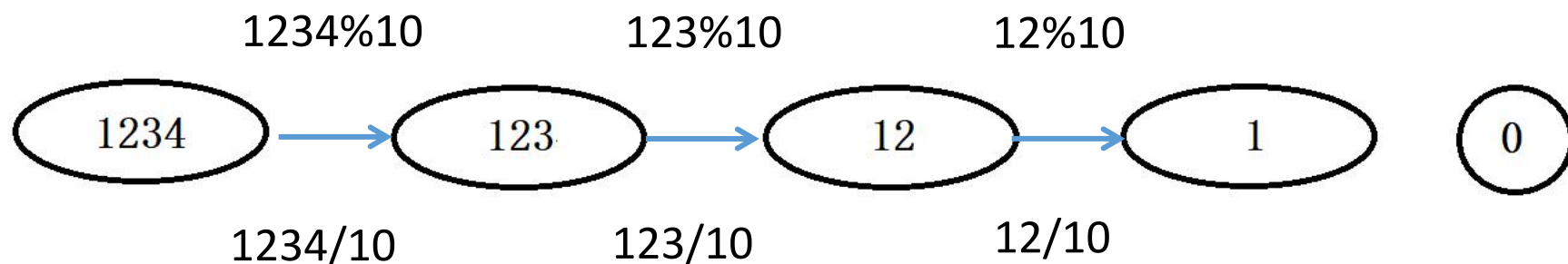
total=1534

[2739]猴子吃桃问题

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int n;
4  int f(int m){
5      if(m==n)
6          return 1;
7      else
8          return 2*(f(m+1)+1);
9  }
10 int main(){
11     cin>>n;
12     printf("total=%d",f(1));
13     return 0;
14 }
```

[2208] 求一个整数的各位数字之和

输入一个整数，求它的各位数字之和。



$$f(n) = f(n/10) + n \% 10$$

[2208] 求一个整数的各位数字之和

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int f(int n){
4      if(n==0)
5          return 0;
6      else
7          return n%10+f(n/10);
8  }
9  int main(){
10     int n;
11     cin>>n;
12     cout<<f(n);
13     return 0;
14 }
```

```

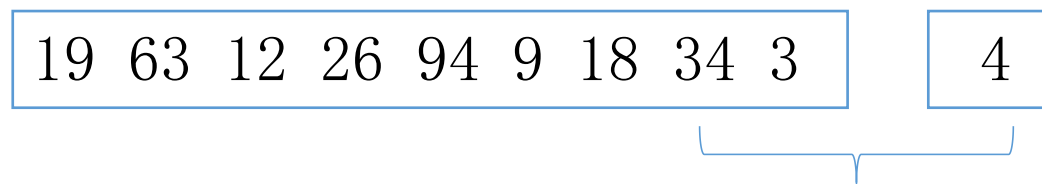
1  #include<bits/stdc++.h>
2  using namespace std;
3  int f(int n){
4      if(n==__1__)
5          return __2__;
6      else
7          return __3__;
8  }
9  int main(){
10     int n;
11     cin>>n;
12     cout<<f(n);
13     return 0;
14 }

```

[2198] 求最小值

输入10个整数，求出其中的最小值。

19 63 12 26 94 9 18 34 3 4



$$f(n) = \min(f(n-1), a[n])$$

[2198] 求最小值

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int a[11];
4  int fmin(int a[],int i)
5  {   if (i==1)
6      return a[1];
7      else
8      return min(fmin(a,i-1),a[i]);
9  }
10
11 int main(){
12     int i,min;
13     for(i=1;i<=10;i++) scanf("%d",&a[i]);
14     min=fmin(a,10);
15     printf("%d",min);
16     return 0;
17 }
```

[2198] 求最小值

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int a[11];
4  int fmin(int a[],int i)
5  {   if (i==__1__)
6      return __2__ ;
7      else
8      return __3__
9  }
10
11 int main(){
12     int i,min;
13     for(i=1;i<=10;i++) scanf("%d",&a[i]);
14     min=fmin(a,10);
15     printf("%d",min);
16     return 0;
17 }
```

[2460] 最大公约数

思考：在初等数学里是如何求两个数的最大公约数？

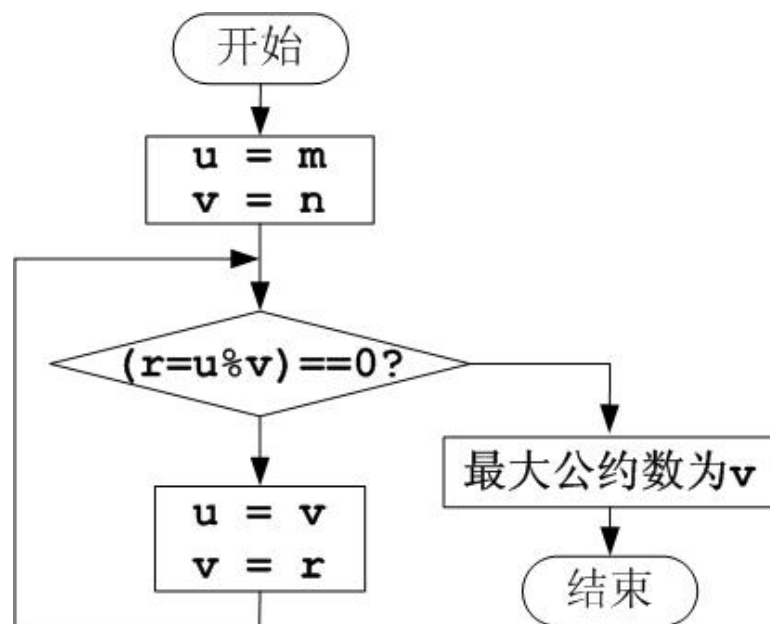
分析：

数论中有一个求最大公约数的算法称为**辗转相除法**，又称**欧几里德(Euclid)算法**。其基本思想及执行过程为(设 m 为两正整数中较大者， n 为较小者)：

(1) 令 $u = m$ ， $v = n$ ；

(2) 取 u 对 v 的余数，即 $r = u \% v$ ，如果 r 的值为0，则此时 v 的值就是 m 和 n 的最大公约数，否则执行第(3)步；

(3) $u = v$ ， $v = r$ ，即 u 的值为 v 的值，而 v 的值为余数 r 。并转向第(2)步。

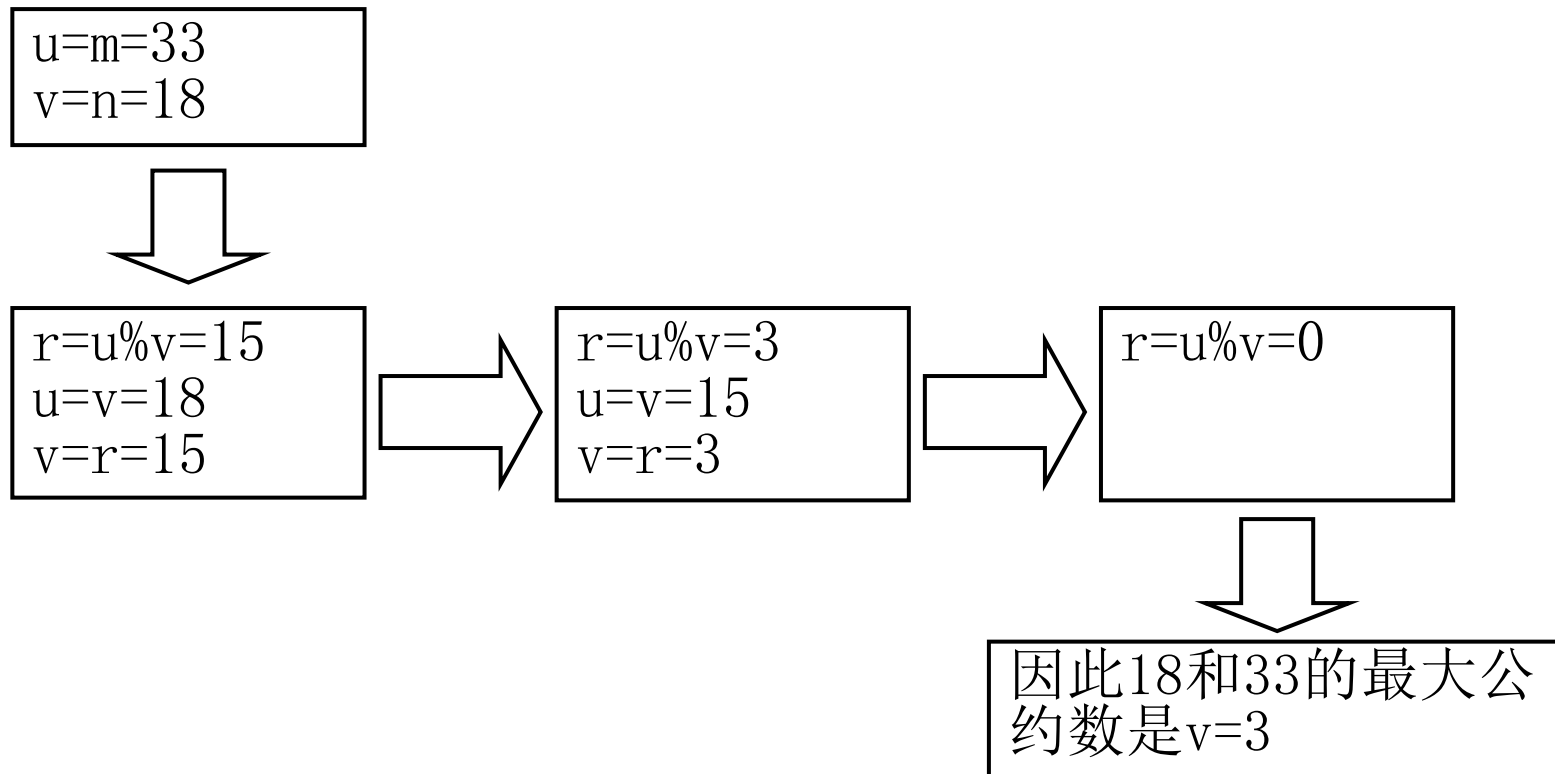


辗转相除法的流程图

[2460] 最大公约数

假设输入的两个正整数为18和33，则 $m = 33$ ， $n = 18$ 。

辗转相除法求最大公约数的过程如下图所示。

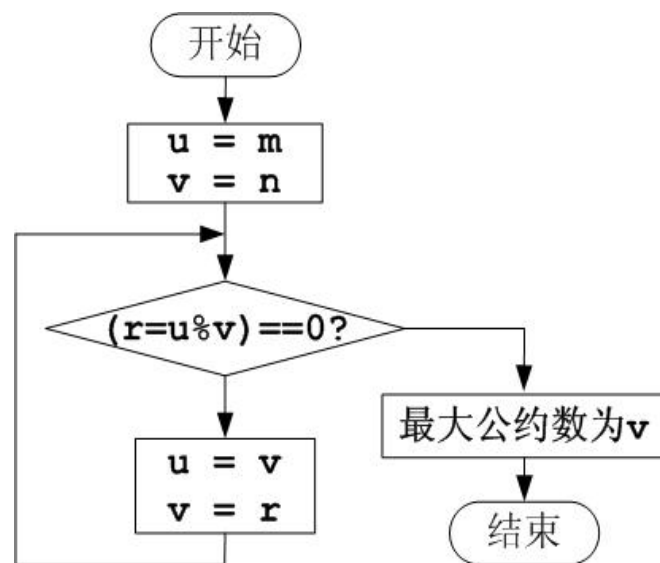


[2460] 最大公约数

辗转相除法的思想也可以采用**递归方法**实现。其递归思想是：在求最大公约数过程中，如果 u 对 v 取余的结果为0，则最大公约数就是 v ；否则**递归**求 v 和 $u\%v$ 的最大公约数。因此，上述代码中的gcd函数可改写成：

```
int gcd( int u, int v ) //求u和v的最大公约数
{
    if( u%v==0 ) return v;
    else return gcd(v, u%v);
}
```

在使用上述递归gcd函数求 “gcd(33,18)”时，要递归调用 “gcd(18,15)”；在执行 “gcd(18,15)”时又递归调用 “gcd(15,3)”；而在执行 “gcd(15,3)”时，因为 $15\%3$ 的结果为0，所以最终求得的最大公约数为3。



辗转相除法的流程图

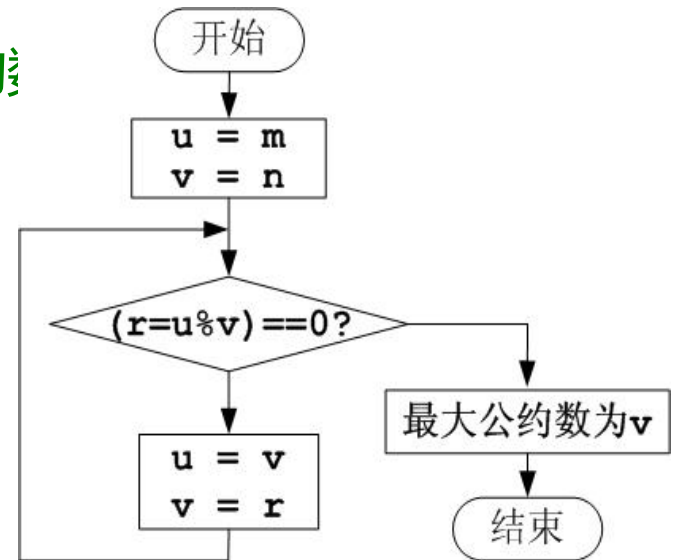
[2460] 最大公约数

```
int gcd( int u, int v )    //求u和v的最大公约数
```

```
{  
    int r;  
    while( (r=u%v)!=0 )  
    { u=v; v=r; }  
    return(v);  
}
```

```
int main( )
```

```
{  
    int m, n, t;  
    scanf( "%d%d", &m, &n );  
    if( m<n )    //交换m和n, 使得m为二者较大者  
    { t = m; m = n; n = t; };  
    printf( "%d.\n", gcd(m,n) );  
    return 0;  
}
```



辗转相除法的流程图

//交换m和n, 使得m为二者较大者



[2646]角谷猜想

角谷猜想，是指对于任意一个正整数，如果是奇数，则乘3加1，如果是偶数，则除以2，得到的结果再按照上述规则重复处理，最终总能够得到1。如，假定初始整数为5，计算过程分别为16、8、4、2、1。程序要求输入一个整数，将经过处理得到1的过程输出来。

$$5*3+1=16$$

$$16/2=8$$

$$8/2=4$$

$$4/2=2$$

$$2/2=1$$

End

递归式：

$$\text{奇数} \quad f(n)=f(n*3+1)$$

$$\text{偶数} \quad f(n)=f(n/2)$$

退出条件 $n=1$

分析 n=5

```
int main(){  
    f1(5);  
}
```



```
3 void f1(int n){  
4     if(n%2){  
5         cout<<n<<"*3+1="<<n*3+1<<"\n";  
6         f2(n*3+1);  
7     }  
8     else{  
9         cout<<n<<"/2="<<n/2<<"\n";  
10        f2(n/2);  
11    }  
12 }
```

n=16

```
3 void f2(int n){  
4     if(n%2==0){  
5         cout<<n<<"/2="<<n/2<<"\n";  
6         f3(n/2);  
7     }  
8 }
```

n=8

```
3 void f3(int n){  
4     if(n%2==0){  
5         cout<<n<<"/2="<<n/2<<"\n";  
6         f4(n/2);  
7     }  
8 }
```

n=4

```
3 void f5(int n){  
4     if(n==1)  
5     {  
6         cout<<"End";  
7         return;  
8     }  
9 }
```

n=1

```
3 void f4(int n){  
4     if(n%2==0){  
5         cout<<n<<"/2="<<n/2<<"\n";  
6         f5(n/2);  
7     }  
8 }
```

n=2

```
3 void f(int n){
4     if(n==1)
5     {
6         cout<<"End";
7         return;
8     }
9     if(n%2){
10        cout<<n<<"*3+1="<<n*3+1<<"\n";
11        f(n*3+1);
12    }
13    else{
14        cout<<n<<"/2="<<n/2<<"\n";
15        f(n/2);
16    }
17 }
```

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 void f(int n){
18 int main(){
19     int n;
20     cin>>n;
21     f(n);
22 }
```

[2401] 冒泡法排序

输入10个整数，求出其中的最小值。

19 63 12 26 94 9 18 34 3 4

19 12 26 63 9 18 34 3 4

94

12 19 26 9 18 34 3 4

63

12 19 9 18 26 3 4

34

... ..

[2401] 冒泡法排序

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  void f(int a[],int n){
4      if(n==1)
5          return;
6      for(int i=1;i<n;i++){
7          if(a[i]>a[i+1])
8              swap(a[i],a[i+1]);
9      }
10     f(a,n-1);
11 }
```

```
12 int main(){
13     int a[11];
14     for(int i=1;i<=10;i++){
15         cin>>a[i];
16     }
17     f(a,10);
18     for(int i=1;i<=10;i++){
19         cout<<a[i]<<" ";
20     }
21     return 0;
22 }
```

[3044] 最大公约数和最小公倍数

输入二个正整数 x_0 , y_0 , 求出满足下列条件的 P , Q 的个数
条件:

1. P , A 是正整数;
2. 要求 P , Q 以 x_0 为最大公约数, 以 y_0 为最小公倍数。

求满足条件的所有可能的两个正整数的个数。

每个测试文件只包含一组测试数据, 每组两个正整数 x_0 和 y_0 ($2 \leq x_0 < 100000$, $2 \leq y_0 < 1000000$)。

对于每组输入数据, 输出满足条件的所有可能的两个正整数的个数。

[3044] 最大公约数和最小公倍数

样例数据的说明：

输入3 60

此时的P Q分别为：

3 60

15 12

12 15

60 3

所以，满足条件的所有可能的两个正整数的个数共4种。

样例输入

3 60

样例输出

4

P, Q的最大公约数为 x_0 , 即
 $\gcd(P, Q) = x_0$ ($\gcd$ 为求最大公约数的函数), 并且
 $P*Q/\gcd(P, Q) = y_0$,
显然,

$$P*Q = x_0*y_0$$

可以看做

$$P = x_0*y_0/Q,$$

同时要满足P, Q为正整数, 并且

$$\gcd(P, Q) = x_0$$

```

1  #include<iostream>
2  using namespace std;
3  int gcd(int a, int b)
4  {
5      return b == 0 ? a : gcd(b, a % b);
6  } //用于求a和b的最大公约数
7  int main()
8  {
9      int n, m, sum;
10     cin >> n >> m;
11     sum = n * m;
12     int num = 0;
13     for (int i = 1; i <= sum; i++)
14     {
15         if (sum % i == 0)
16         {
17             if (gcd(i, sum / i) == n)
18                 num++; //当同时满足P和Q的最大公约数为n时, 即P,Q满足条件
19             //判断P和Q是否都为整数, 并且P*Q=sum=n*m
20         }
21     }
22     cout << num << endl;

```

1 递归理解

一般来说，能够用递归解决的问题应该满足以下三个条件：

- 需要解决的问题可以转化为一个或多个子问题来求解，而这些子问题的求解方法与原问题完全相同，只是在数量规模上不同。
- 递归调用的次数必须是有限的。
- 必须有结束递归的条件来终止递归。



2 何时使用递归

在以下三种情况下，常常要用到递归的方法。

1. 定义是递归的

有许多数学公式、数列等的定义是递归的。例如，求 $n!$ 和Fibonacci数列等。这些问题的求解过程可以将其递归定义直接转化为对应的递归算法。



1.1 何时使用递归

2. 数据结构是递归的

有些数据结构是递归的。例如单链表就是一种递归数据结构，其结点类型声明如下：

```
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
} LinkList;
```

结构体LNode的定义中用到了它自身，即指针域next是一种指向自身类型的指针，所以它是一种递归数据结构。



1.1 何时使用递归

对于递归数据结构，采用递归的方法编写算法既方便又有效。例如，求一个不带头结点的单链表L的所有data域（假设为int型）之和的递归算法如下：

```
int Sum(LinkList *L)
{   if (L==NULL)
        return 0;
    else
        return(L->data+Sum(L->next));
}
```



例1：递归求和

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  int a[11]={0,1,2,3,4,5,6,7,8,9,10};
4  int sum(int n){
5      if(n==0)return 0;
6      else return a[n]+sum(n-1);
7  }
8  }
9  int main()
10 {
11     cout<<sum(10)<<endl;
12 }
```

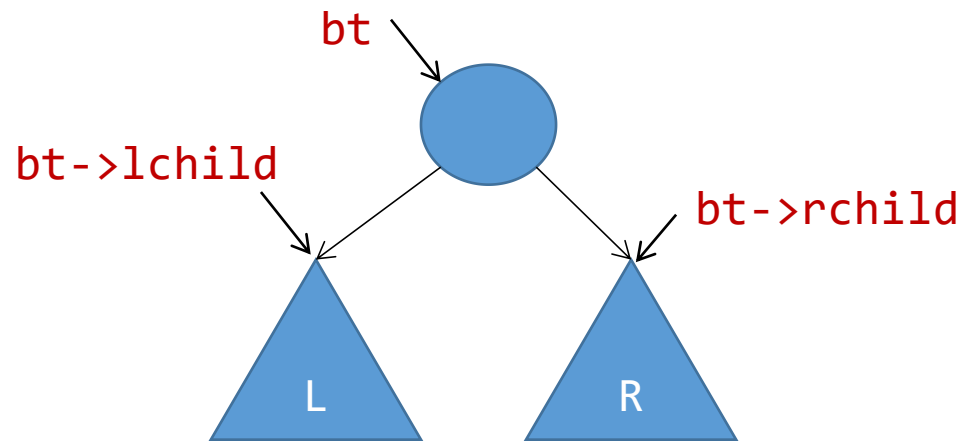
```
5  int sum(int n){
6      cout<<s<<endl;
7      if(n==0)s=0;
8      else s=a[n]+sum(n-1);
9      return s;
10 }
```

例2：二叉树求和

设计求非空二叉链bt中所有结点值之和的递归算法，
假设二叉链的data域为int型。

```
typedef struct BNode
{
    int data;
    struct BNode *lchild, *rchild;
} BTreeNode;           //二叉链结点类型
```





```
int Sumbt(BTNode *bt)           //求二叉树bt中所有结点值之和
{  if (bt->lchild==NULL && bt->rchild==NULL)
    return bt->data;             //只有一个结点时返回该结点值
    else                          //否则返回左、右子树结点值之和加上根结点值
        return Sumbt(bt->lchild)+ Sumbt(bt->rchild)+bt->data);
}
```



1.1 何时使用递归

3. 问题的求解方法是递归的

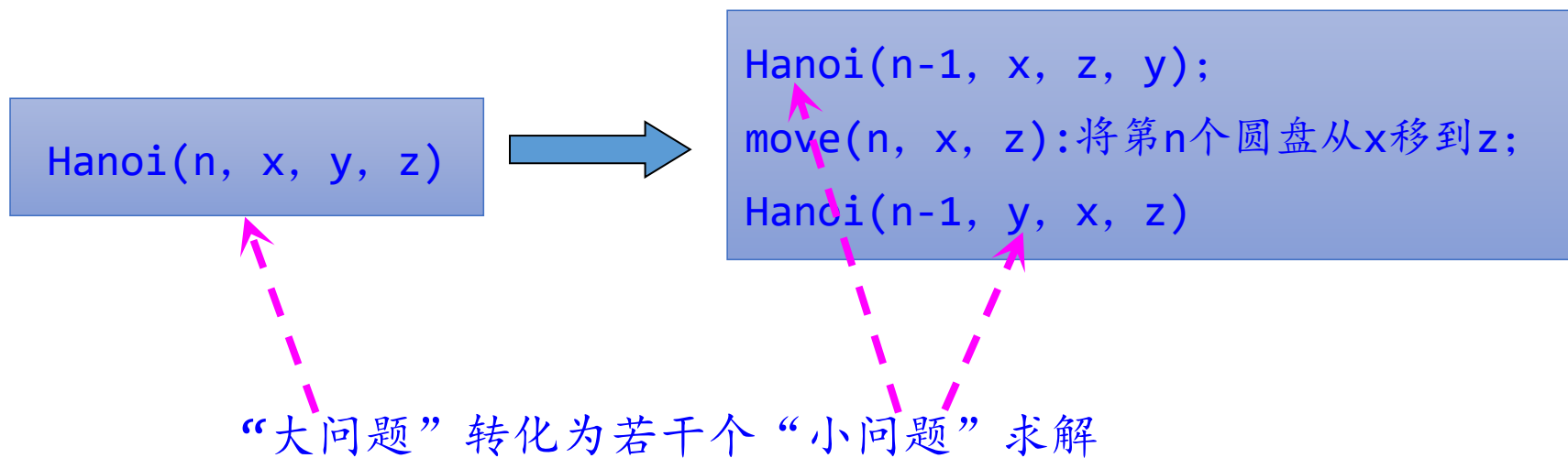
有些问题的解法是递归的，典型的有Hanoi问题求解。



盘片移动时必须遵守以下规则：每次只能移动一个盘片；盘片可以插在X、Y和Z中任一塔座；任何时候都不能将一个较大的盘片放在较小的盘片上。



设 $\text{Hanoi}(n, x, y, z)$ 表示将 n 个盘片从 x 通过 y 移动到 z 上，
递归分解的过程是：



1.2 递归模型

递归模型是递归算法的抽象，它反映一个递归问题的递归结构。例如前面的递归算法对应的递归模型如下：

```
fun(1)=1                                (1)
fun(n)=n*fun(n-1)    n>1    (2)
```

其中，第一个式子给出了递归的终止条件，第二个式子给出了 $\text{fun}(n)$ 的值与 $\text{fun}(n-1)$ 的值之间的关系，我们把第一个式子称为递归出口，把第二个式子称为递归体。



一般地，一个递归模型是由递归出口和递归体两部分组成，前者确定递归到何时结束，后者确定递归求解时的递推关系。

递归出口的一般格式如下：

$$f(s_1)=m_1$$

这里的 s_1 与 m_1 均为常量，有些递归问题可能有几个递归出口。

递归体的一般格式如下：

$$f(s_{n+1})=g(f(s_i), f(s_{i+1}), \dots, f(s_n), c_j, c_{j+1}, \dots, c_m) \quad (2.2)$$

其中， n 、 i 、 j 和 m 均为正整数。这里的 s_{n+1} 是一个递归“大问题”， s_i 、 s_{i+1} 、 $\dots$ 、 s_n 为递归“小问题”，

1.2.1 递归算法的执行过程

- 一个正确的递归程序虽然每次调用的是相同的子程序，但它的参量、输入数据等均有变化。
 - 在正常的情况下，随着调用的不断深入，必定会出现调用到某一层的函数时，不再执行递归调用而终止函数的执行，遇到递归出口便是这种情况。
-
- 递归调用是函数嵌套调用的一种特殊情况，即它是调用自身代码。也可以把每一次递归调用理解成调用自身代码的一个复制件。
 - 由于每次调用时，它的参量和局部变量均不相同，因而也就保证了各个复制件执行时的独立性。

1.2.1 递归算法的执行过程

- 系统为每一次调用开辟一组存储单元，用来存放本次调用的返回地址以及被中断的函数的参量值。
- 这些单元以系统栈的形式存放，每调用一次进栈一次，当返回时执行出栈操作，把当前栈顶保留的值送回相应的参量中进行恢复，并按栈顶中的返回地址，从断点继续执行。



求5!

执行fun(5)时内部
栈的变化及求解过程
如下：

```
void main()
{  printf("%d\n", fun(5));  }
```

fun(5)调用：进栈



5	fun(4)*5
---	----------

n 函数值

fun(4)调用：进栈



4	fun(3)*4
5	fun(4)*5

fun(3)调用：进栈



3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

fun(2)调用：进栈



2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

fun(1)调用：进栈并求值



1	1
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

退栈1次并求fun(2)值



2	1*2 = 2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5



退栈1次并求fun(3)值



3	$2*3=6$
4	$\text{fun}(3)*4$
5	$\text{fun}(4)*5$

退栈1次并求fun(4)值



4	$6*4=24$
5	$\text{fun}(4)*5$

退栈1次并求fun(5)值



5	$24*5=120$
---	------------

退栈1次并输出120



1.2.1 递归算法的执行过程

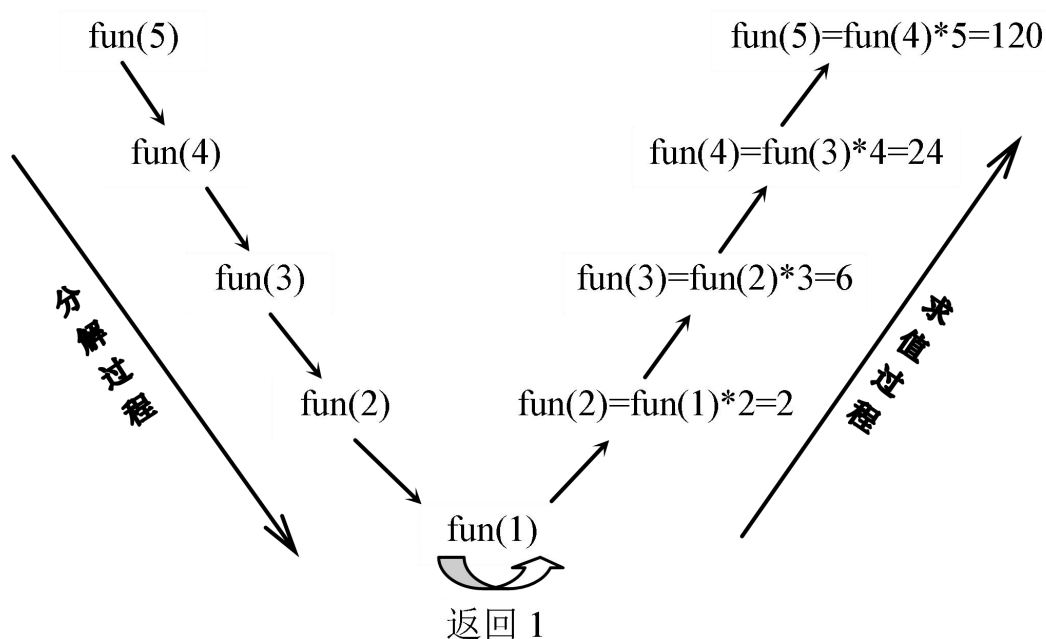
从以上过程可以得出：

- 每递归调用一次，就需进栈一次，最多的进栈元素个数称为递归深度，当 n 越大，递归深度越深，开辟的栈空间也越大。
- 每当遇到递归出口或完成本次执行时，需退栈一次，并恢复参量值，当全部执行完毕时，栈应为空。



1.2.1 递归算法的执行过程

归纳起来，递归调用的实现是分两步进行的，第一步是分解过程，即用递归体将“大问题”分解成“小问题”，直到递归出口为止，然后进行第二步的求值过程，即已知“小问题”，计算“大问题”。前面的 $\text{fun}(5)$ 求解过程如下所示。



Fibonacci数列定义为：

$\text{Fib}(n)=1$ $n=1$

$\text{Fib}(n)=1$ $n=2$

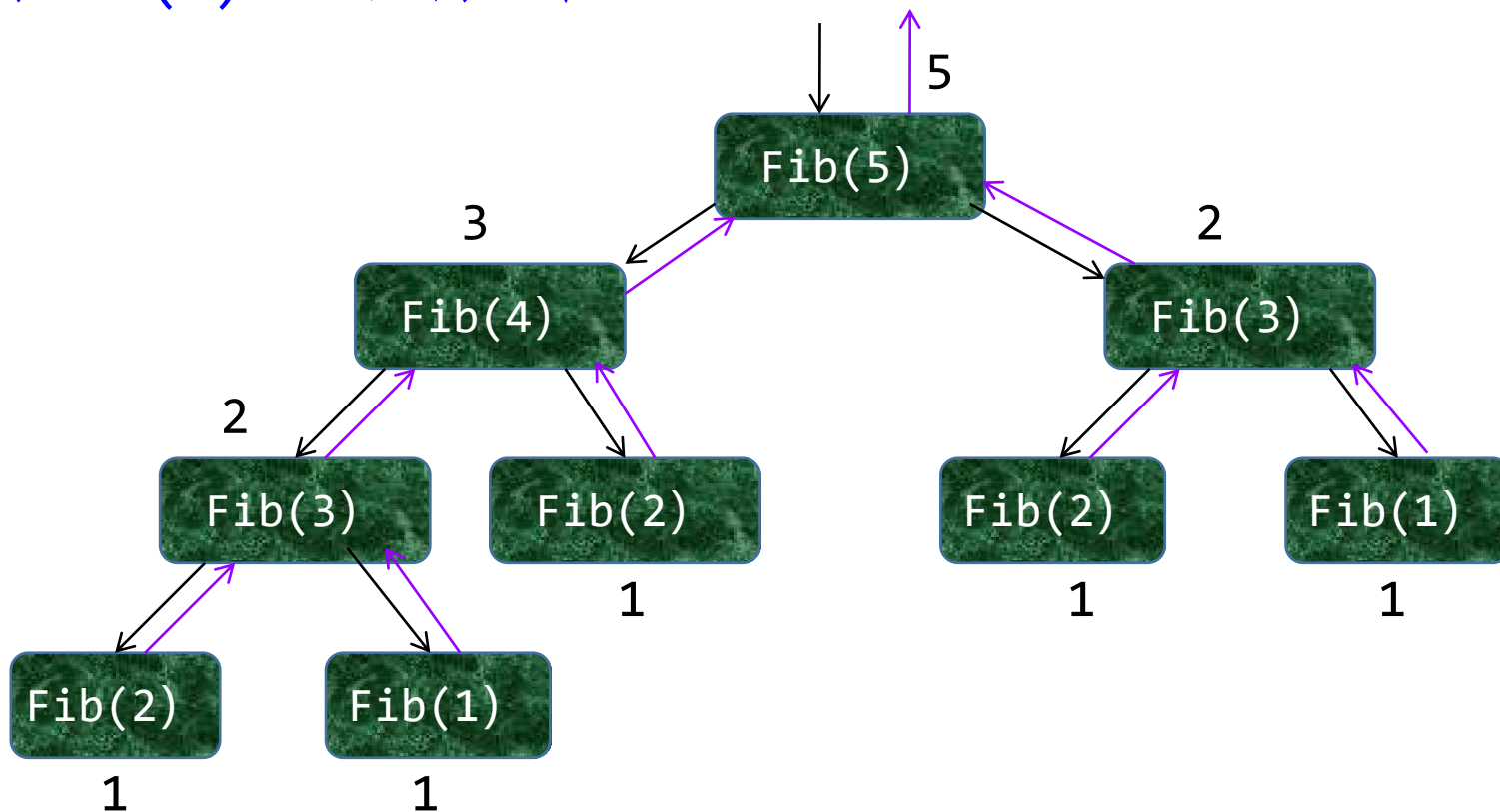
$\text{Fib}(n)=\text{Fib}(n-1)+\text{Fib}(n-2)$ $n>2$

对应的递归算法如下：

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



求Fib(5)的递归树如下：



从上面求Fib(5)的过程看到，对于复杂的递归调用，分解和求值可能交替进行、循环反复，直到求出最终值。

-
- 在递归函数执行时，形参会随着递归调用发生变化，但每次调用后会恢复为调用前的形参，将递归函数的非引用型形参的取值称为状态。
 - 递归函数的引用型形参在执行后会回传给实参，有时类似全局变量，不作为状态的一部分，在调用过程中状态会发生变化，而一次调用后会自动恢复为调用前的状态。



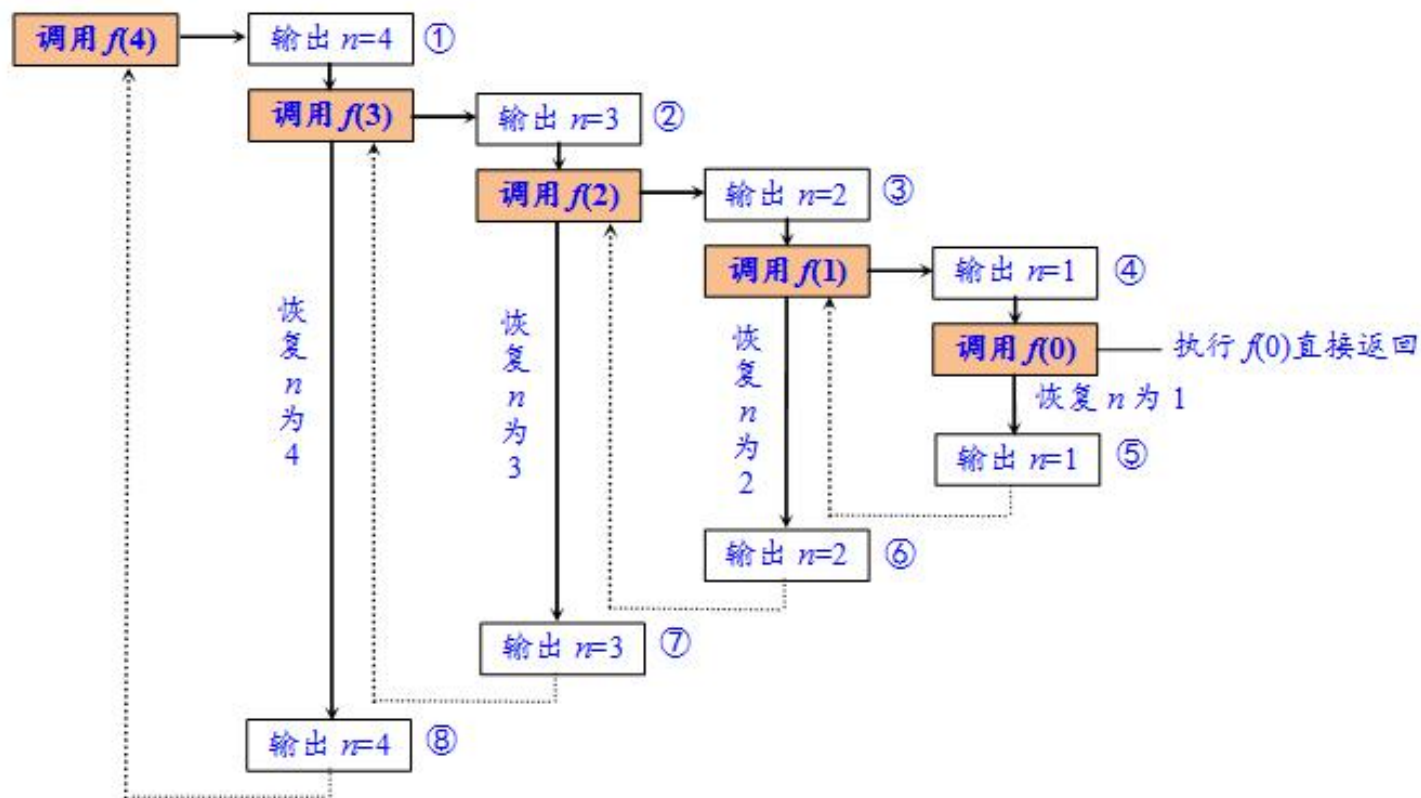
例如，有以下递归程序：

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  int f(int n)
4  {
5      if (n<1) return 1;
6      else
7      {
8          printf("调用f(%d)前, n=%d\n",n-1,n);
9          f(n-1);
10         printf("调用f(%d)后:n=%d\n",n-1,n);
11     }
12 }
13 int main(){
14     cout<<f(5);
15     return 0;
16 }
```

执行结果

调用f(3)前, n=4
调用f(2)前, n=3
调用f(1)前, n=2
调用f(0)前, n=1
调用f(0)后: n=1
调用f(1)后: n=2
调用f(2)后: n=3
调用f(3)后: n=4

在上述递归函数 f 中，状态为 (n) ，其递归执行过程如图所示，输出框旁的数字表示输出顺序，虚线表示本次递归调用执行完后返回，从中看到每次递归调用后状态都恢复为调用前的状态。



1.2.2 递归算法设计步骤

递归算法设计先要给出递归模型，再转换成对应的C/C++语言函数。

(1) 对原问题 $f(s_n)$ 进行分析，抽象出合理的“小问题” $f(s_{n-1})$ （与数学归纳法中假设 $n=k-1$ 时等式成立相似）；

(2) 假设 $f(s_{n-1})$ 是可解的，在此基础上确定 $f(s_n)$ 的解，即给出 $f(s_n)$ 与 $f(s_{n-1})$ 之间的关系（与数学归纳法中求证 $n=k$ 时等式成立的过程相似）；

(3) 确定一个特定情况（如 $f(1)$ 或 $f(0)$ ）的解，由此作为递归出口（与数学归纳法中求证 $n=1$ 或 $n=0$ 时等式成立相似）。

例3：用递归法求一个整数数组a的最大元素。

解：设 $f(a, i)$ 求解数组 a 中前 i 个元素即 $a[0..i-1]$ 中的最大元素，则 $f(a, i-1)$ 求解数组 a 中前 $i-1$ 个元素即 $a[0..i-2]$ 中的最大元素，前者为“大问题”，后者为“小问题”。

假设 $f(a, i-1)$ 已求出，则有 $f(a, i) = \text{MAX}\{f(a, i-1), a[i-1]\}$ 。递推方向是朝 a 中元素减少的方向推进，当 a 中只有一个元素时，该元素就是最大元素，所以 $f(a, 1) = a[0]$ 。



由此得到递归模型如下：

$f(a, i) = a[0]$	当 $i=1$ 时
$f(a, i) = \text{MAX}\{f(a, i-1), a[i-1]\}$	当 $i > 1$ 时

对应的递归算法如下：

```
int fmax(int a[], int i)
{
    if (i==1)
        return a[0];
    else
        return(fmax(a, i-1), a[i-1]);
}
```



例4：简单选择排序和冒泡排序

【问题描述】 对于给定的含有 n 个元素的数组 a ，分别采用简单选择排序和冒泡排序方法对其按元素值递增排序。



1. 简单选择排序

设 $f(a, n, i)$ 用于对 $a[i..n-1]$ 元素序列（共 $n-i$ 个元素）进行简单选择排序，是“大问题”。

$f(a, n, i+1)$ 用于对 $a[i+1..n-1]$ 元素序列（共 $n-i-1$ 个元素）进行简单选择排序，是“小问题”。

当 $i=n-1$ 时所有元素有序，算法结束。

$f(a, n, i) \equiv$ 不做任何事情，算法结束

当 $i=n-1$

$f(a, n, i) \equiv$ 通过简单比较挑选 $a[i..n-1]$ 中的最

小元素 $a[k]$ 放在 $a[i]$ 处；

否则

$f(a, n, i+1);$



```
void SelectSort(int a[], int n, int i)
{   int j, k;
    if (i==n-1) return;           //满足递归出口条件

    else
    {   k=i;                       //k记录a[i..n-1]中最小元素的下标
        for (j=i+1;j<n;j++)       //在a[i..n-1]中找最小元素
            if (a[j]<a[k])
                k=j;
        if (k!=i)                 //若最小元素不是a[i]
            swap(a[i],a[k]);       //a[i]和a[k]交换

        SelectSort(a, n, i+1);
    }
}
```



2. 冒泡排序

设 $f(a, n, i)$ 用于对 $a[i..n-1]$ 元素序列（共 $n-i$ 个元素）进行冒泡排序，是“大问题”，则 $f(a, n, i+1)$ 用于对 $a[i+1..n-1]$ 元素序列（共 $n-i-1$ 个元素）进行冒泡排序，是“小问题”。当 $i=n-1$ 时所有元素有序，算法结束。

$f(a, n, i) \equiv$ 不做任何事情，算法结束

当 $i=n-1$

$f(a, n, i) \equiv$ 对 $a[i..n-1]$ 元素序列，从 $a[n-1]$ 开始
进行相邻元素比较；

否则

若相邻两元素反序则将两者交换；

若没有交换则返回，否则执行 $f(a, n, i+1)$ ；

```
void BubbleSort(int a[], int n, int i)
{
    int j;
    bool exchange;
    if (i==n-1) return;           //满足递归出口条件

    else
    {
        exchange=false;          //置exchange为false
        for (j=n-1;j>i;j--)
            if (a[j]<a[j-1])       //当相邻元素反序时
            {
                swap(a[j],a[j-1]); //发生交换置exchange为true
                exchange=true;
            }
        if (exchange==false)      //未发生交换时直接返回
            return;
        else                       //发生交换时继续递归调用
            BubbleSort(a, n, i+1);
    }
}
```



1.2.3[2749]汉诺塔问题

约19世纪末，在欧洲的商店中出售一种智力玩具，在一块铜板上有三根杆，最左边的杆上自上而下、由小到大顺序串着由64个圆盘构成的塔。目的是将最左边杆上的盘全部移到中间的杆上，条件是一次只能移动一个盘，且不允许大盘放在小盘的上面。

这是一个著名的问题，几乎所有的教材上都有这个问题。由于条件是一次只能移动一个盘，且不允许大盘放在小盘上面，所以64个盘的移动次数是：18,446,744,073,709,551,615这是一个天文数字，若每一微秒可能计算(并不输出)一次移动，那么也需要几乎一百万年。我们仅能找出问题的解决方法并解决较小N值时的汉诺塔，但很难用计算机解决64层的汉诺塔。假定圆盘从小到大编号为1, 2, ...

[2749]汉诺塔问题

输入

输入为一个整数(小于20) 后面跟三个单字符串。
整数为盘子的数目，后三个字符表示三个杆子的编号。

输出

输出每一步移动盘子的记录。一次移动一行。

每次移动的记录为例如 a->3->b 的形式，即把编号为3的盘子从a杆移至b杆。

样例输入

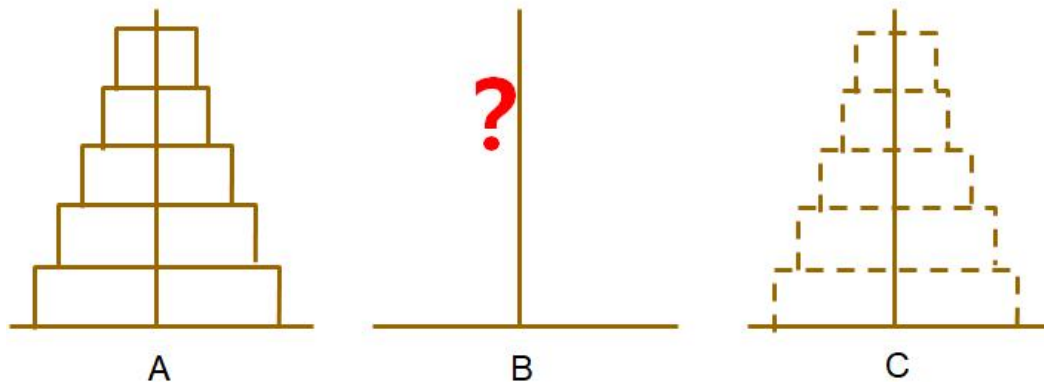
2 a b c

样例输出

a->1->c

a->2->b

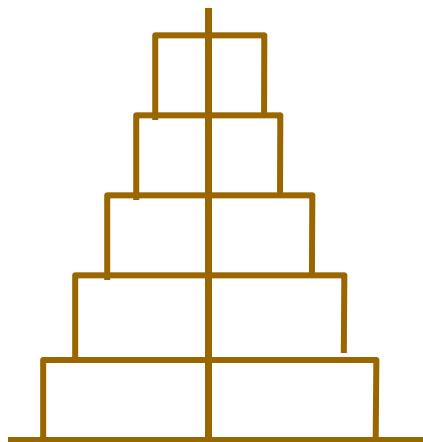
c->1->b



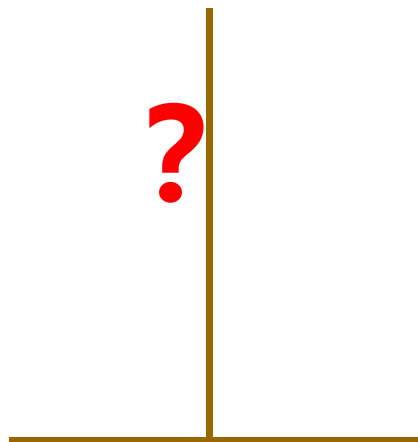
分析



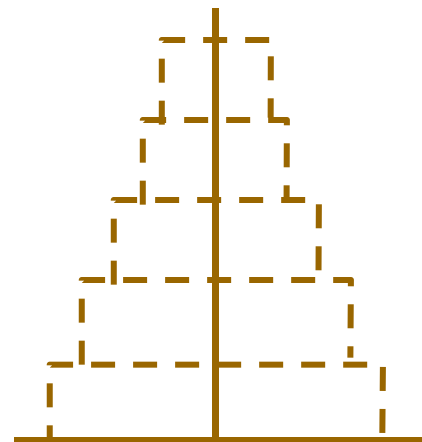
塔内有A, B, C三个柱子。起初, A柱上有 n 个盘子, 依次由大至小、从下往上堆放; **要求把它们全部移到B柱上**, 在移动过程中可以利用C柱, 但**每次只能移动一个盘子**, 且**必须使三个座始终保持大盘在下, 小盘在上的状态**。



A



B



C



分析(按A->C)

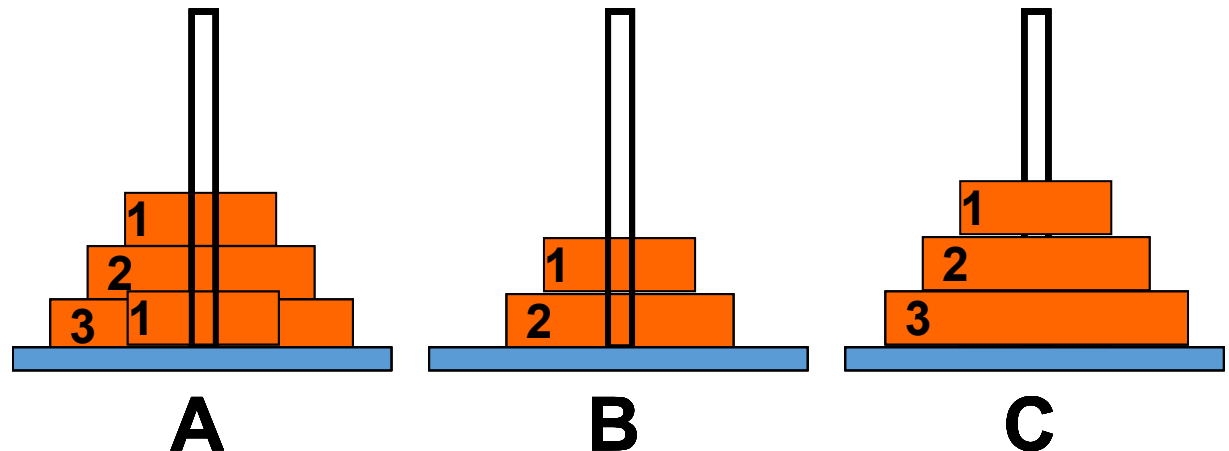
如果盘子数量很少，通过心算便可以直接想出移动盘子的具体步骤。

❖ 比如：只有 2 个盘子的情况，此问题就变得相当的简单，只需要 3 步就可以完成整个移动操作。

- A—>B (表示将 A柱 最上面的 1 个盘子移到 B柱 上)
- A—>C (将 A 此时最上面的 1 个盘子移到 C 上)
- B—>C (将 B 上的盘子移到 C 上)

❖ 又如：移动 3 个盘子的情况，需要 7 步

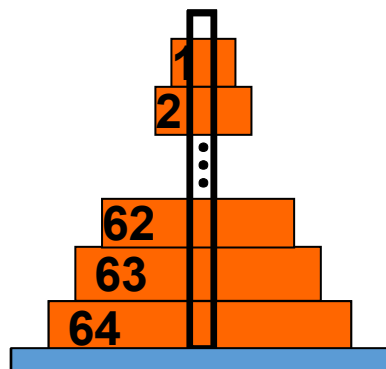
- A—>C
- A—>B
- C—>B
- A—>C
- B—>A
- B—>C
- A—>C



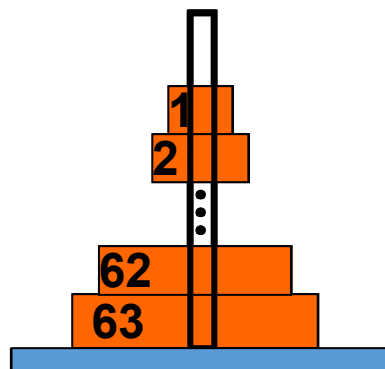
分析

思考： n 个盘子至少需要移动多少步？比如 $n=64$ 。

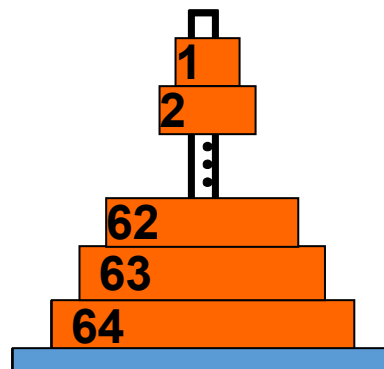
现在我想不出移完64个盘子的步骤，但如果另有一个人能想出怎样移完63个盘子，我只要移最后1个盘子就可以了！



A



B



C

第一个人移动64个盘子(A→C)的过程：

1. 命令第2个人将63个盘子从A移到B上；
2. 自己将剩余的最底下最大的那1个盘子从A移到C上；
3. 再命令第2个人将63个盘子从B移到C上；
4. 完成任务！

第2个人移动63个盘子(A→B)的过程：

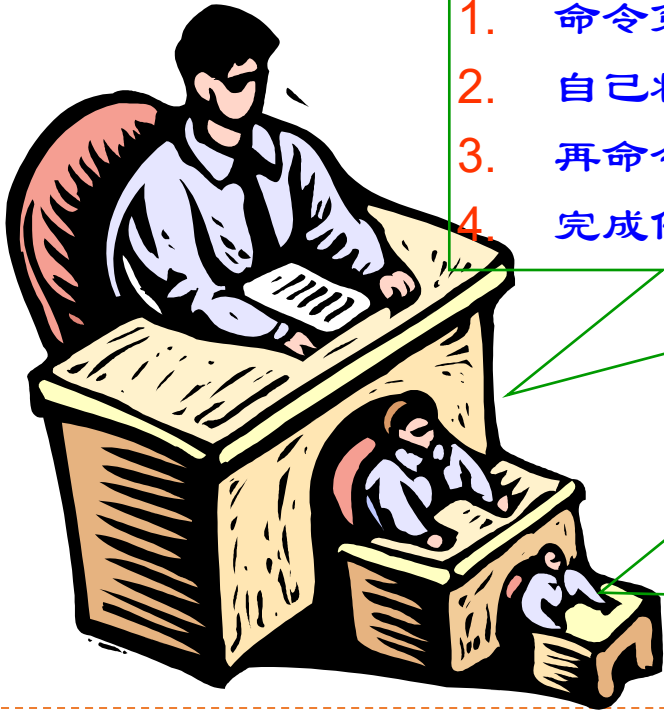
1. 命令第3个人将62个盘子从A移到C上；
2. 自己将剩余的最底下最大的那1个盘子从A移到B上；
3. 再命令第3个人将62个盘子从C移到B上；
4. 完成任务！

第3个人移动62个盘子(A→C)的过程：

1. 命令第4个人将61个盘子从A移到B上；
2. 自己将剩余的最底下最大的那1个盘子从A移到C上；
3. 再命令第4个人将61个盘子从B移到C上；
4. 完成任务！

.....

层层下放！递归调用！





思考：递归的结束条件是什么？

❖ 递归的结束条件：

- 最后 1 个人（第 64 个人）只需要移动 1 个盘子。

❖ 注意：

- 第 1 个人完成任务的前提是：第 2 个人完成了任务
- 第 2 个人完成任务的前提是：第 3 个人完成了任务
-
- 第 63 个人完成任务的前提是：第 64 个人完成了任务

❖ 所以：

- 只有当第 64 个人完成任务后，第 63 个人才能完成任务；只有第 2~64 个人完成任务后，第 1 个人才能完成任务！

典型的递归问题！

❖ 将 n 个盘子从 A 移到 C 可以分解为以下 3 个步骤：

- 1) 将 A 上 $n-1$ 个盘子借助 C 先移到 B 上；
- 2) 把 A 上剩下的 1 个盘移到 C 上；
- 3) 将 $n-1$ 个盘从 B 借助 A 移到 C 上。

❖ 上面第 1 步和第 3 步，都是把 $n-1$ 个盘子从 1 个座移到另 1 个座上，采用的办法是相同的，只是座的名字不同而已。为使之一般化，可以将第 1 步和第 3 步表示为：

- 将 one 座上 $n-1$ 个盘子移到 two 座（借助 three 座）。
- 只是在第 1 步和第 3 步中，one、two、three 和 A、B、C 的对应关系不同。
- 对第 1 步，对应关系是：one—A，two—B，three—C。
- 对第 3 步，对应关系是：one—B，two—C，three—A。

❖ 因此，将上面 3 个步骤分为 2 类操作：

- 将 $n-1$ 个盘子从 1 个座移到另 1 个座上（当 $n>1$ 时）；
- 将最后 1 个盘子从 1 个座上移到另 1 个座上。

★ 设计程序

❖ 分别用 2 个函数实现以上 2 类操作：

- 设计 hanoi 函数实现第 1 类操作；
 - ◆ 函数 `hanoi( n, one, two, three )` 将实现把 n 个盘子从 one 座借助 two 座移到 three 座的过程；
- 设计 move 函数实现第 2 类操作；
 - ◆ 函数 `move( x, y )` 将实现把 1 个盘子从 x 座移到 y 座的过程。 x 和 y 是代表 A、B、C 座之一，根据每次不同情况分别以 A、B、C 代入。



```

1  #include<bits/stdc++.h>
2  using namespace std;
3      //出发  终点      中间
4  void hanTa(int n,char one,char two,char three){
5      if(n==1)printf("%c->%d->%c\n",one,n,two);
6      else{
7          hanTa(n-1,one,three,two);
8          printf("%c->%d->%c\n",one,n,two);
9          hanTa(n-1,three,two,one);
10     }
11
12 }
13 int main(){
14     int n;
15     char a,b,c;
16     cin>>n>>a>>b>>c;
17     hanTa(n,a,b,c);
18     return 0;
19 }

```

1.2.4[2764] 放苹果

把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用K表示）5, 1, 1和1, 5, 1是同一种分法。

输入

第一行是测试数据的数目t ($0 \leq t \leq 20$)。以下每行均包含二个整数M和N，以空格分开。 $1 \leq M, N \leq 10$ 。

输出

对输入的每组数据M和N，用一行输出相应的K。

样例输入

1

7 3

样例输出

8

分析

这个问题的关键是递推函数。

m 个苹果放在 n 个盘子中，那么定义函数为 $\text{apple}(m,n)$ ：

1. $m=0$ ，没有苹果，那么只有一种放法，即 $\text{apple}(0,n)=1$

2. $n=1$ ，只有一个盘中，不论有或者无苹果，那么只有一种放法，
 $\text{apple}(m,1)=1$

3. $n>m$ ，和 m 个苹果放在 m 个盘子中是一样的，即 $\text{apple}(m,n)=\text{apple}(m,m)$

4. $m>=n$ ，这时分为两种情况，一是所有盘子都有苹果，二是不是所有盘子都有苹果。不是所有盘子都有苹果和至少有一个盘子空着是一样的，即 $=\text{apple}(m,n-1)$ 。所有盘子都有苹果，也就是至少每个盘子有一个苹果， m 个苹果中的 n 个放在 n 个盘子中，剩下的 $m-n$ 个苹果，这和 $m-n$ 个苹果放在 n 个盘子中是一样的，即 $=\text{apple}(m-n, n)$ 。这时， $\text{apple}(m,n)=\text{apple}(m-n, n)+\text{apple}(m,n-1)$ 。

```

1  #include <stdio.h>
2  int apple(int m,int n)
3  {
4      if(m == 0 || n == 1)
5          return 1;
6      else if(n > m)
7          return apple(m, m);
8      else
9          return apple(m - n, n) + apple(m, n - 1);
10 }
11
12 int main(void)
13 {
14     int t, m, n;
15
16     scanf("%d", &t);
17     while(t--) {
18         scanf("%d%d", &m, &n);
19         printf("%d\n", apple(m, n));
20     }
21     return 0;
22 }

```

1.2.5[2867] 集合的划分

设 S 是一个具有 n 个元素的集合， $S = \{a_1, a_2, \dots, a_n\}$ ，现将 S 划分成 k 个满足下列条件的子集合 $S_1, S_2, \dots$ ，且满足：

1. $S_i \neq \emptyset$
2. $S_i \cap S_j = \emptyset$ $(1 \leq i, j \leq k \quad i \neq j)$
3. $S_1 \cup S_2 \cup S_3 \cup \dots \cup S_k = S$

则称 $S_1, S_2, \dots$ ，是集合 S 的一个划分。它相当于把 S 集合中的 n 个元素 $a_1, a_2, \dots, a_n$ 放入 k 个($0 < k \leq n < 30$)无标号的盒子中，使得没有一个盒子为空。请你确定 n 个元素 $a_1, a_2, \dots$ ，放入 k 个无标号盒子中去的划分数 $S(n, k)$ 。

输入

给出 n 和 k 。

输出

n 个元素 $a_1, a_2, \dots, a_n$ 放入 k 个无标号盒子中去的划分数 $S(n, k)$ 。

样例输入10 6

样例输出22827

分析

先举个例子，设 $S = \{1, 2, 3, 4\}$ ， $k=3$ ，不难得出 S 有6种不同的划分方案，即划分数 $S(4, 3)=6$ ，具体方案为：

$\{1, 2\} \cup \{3\} \cup \{4\}$

$\{1, 3\} \cup \{2\} \cup \{4\}$

$\{1, 4\} \cup \{2\} \cup \{3\}$

$\{2, 3\} \cup \{1\} \cup \{4\}$

$\{2, 4\} \cup \{1\} \cup \{3\}$

$\{3, 4\} \cup \{1\} \cup \{2\}$

考虑一般情况，对于任意的含有 n 个元素 $a_1, a_2, \dots, a_n$ 的集合 S ，放入 k 个无标号的盒子中去，划分数为 $S(n, k)$ ，我们很难凭直觉和经验计算划分数和枚举划分的所有方案，必须归纳出问题的本质。

其实对于任一个元素 a_n ，则必然出现以下两种情况：

1、 **$\{a_n\}$ 是 k 个子集中的一个**，于是我们只要把 $a_1, a_2, \dots, a_{n-1}$ 划分为 $k-1$ 子集，便解决了本题，这种情况下的划分数共有

$S(n-1, k-1)$ 个；

2、 **$\{a_n\}$ 不是 k 个子集中的一个**，则 a_n 必与其它的元素构成一个子集。则问题相当于先把 $a_1, a_2, \dots, a_{n-1}$ 划分成 k 个子集，这种情况下划分数共有 $S(n-1, k)$ 个；然后再把元素 a_n 加入到 k 个子集中的任一个中去，共有 k 种加入方式，这样对于 a_n 的每一种加入方式，都可以使集合划分为 k 个子集，因此根据乘法原理，划分数共有

$k * S(n-1, k)$ 个。

综合上述两种情况，应用加法原理，得出 n 个元素的集合 $\{a_1, a_2, \dots, a_n\}$ 划分为 k 个子集的划分数为以下递归公式：

$$S(n, k) = S(n-1, k-1) + k * S(n-1, k) \quad (n > k, k > 0)。$$

下面，我们来确定 $S(n, k)$ 的边界条件.

首先不能把 n 个元素不放进任何一个集合中去，即 $k=0$ 时， $S(n, k)=0$ ；也不可能在不允许空盒的情况下把 n 个元素放进多于 n 的 k 个集合中去，即 $k>n$ 时， $S(n, k)=0$ ；再者，把 n 个元素放进一个集合或把 n 个元素放进 n 个集合，方案数显然都是1，即 $k=1$ 或 $k=n$ 时， $S(n, k)=1$ 。

因此，我们可以得出划分数 $S(n, k)$ 的递归关系式为：

$$S(n, k) = S(n-1, k-1) + k * S(n-1, k) \quad (n > k, k > 0)$$

$$S(n, k) = 0 \quad (n < k) \text{ 或 } (k = 0)$$

$$S(n, k) = 1 \quad (k = 1) \text{ 或 } (k = n)$$

```
1  #include<iostream>
2  using namespace std;
3
4  int s(int n, int k)    //数据还有可能越界, 请用高精度计算
5  {
6      if ((n < k) || (k == 0)) return 0; //满足边界条件, 退出
7      if ((k == 1) || (k == n)) return 1;
8      return s(n-1, k-1) + k * s(n-1, k); //调用下一层递归
9  }
10 int main()
11 {
12     int n, k;
13     cin >> n >> k;
14     cout << s(n, k);
15     return 0;
16 }
```

1.2.6[2745]凑数字

有 n 个数字， $a[1], a[2], a[3], \dots, a[n]$ ，以及一个数字 m 。问 n 个数字中取出一些数字，这些数字的和能否等于 m 。

输入

多组测试数据，读入到文件尾结束。

第一行输入 n, m 。（ $1 \leq n \leq 20, 1 \leq m \leq 100$ ）

第二行输入 n 个数字 $a[1], a[2], a[3], \dots, a[n]$ 。（ $1 \leq a[i] \leq 100$ ）

输出

如果可以，输出YES，否则输出NO。

样例输入

5 5

1 1 1 1 1

5 5

2 2 2 2 2

样例输出

YES

NO

```

1  #include<stdio.h>
2  #include<string.h>
3  int n,m,a[20],flag,sum;
4  void f(int i)
5  {
6
7      if(i>=n)
8          return;
9      sum+=a[i];
10     if(sum==m)
11     {
12         flag=1;
13         return;
14     }
15     f(i+1);
16     sum-=a[i];
17     f(i+1);
18 }

```

```

19 int main()
20 {
21     while(scanf("%d%d",&n,&m)!=EOF)
22     {
23         memset(a,0,sizeof a);
24         int i;
25         for(i=0;i<n;i++)
26             scanf("%d",&a[i]);
27         flag=0,sum=0;
28         f(0);
29         if(flag)
30             printf("YES\n");
31         else
32             printf("NO\n");
33     }
34     return 0;
35 }

```

1.2.7[3804] zb的生日西瓜

今天是阴历七月初五，acm队员zb的生日。zb正在和C小加、never在武汉集训。他想给这两位兄弟买点什么庆祝生日，经过调查，zb发现C小加和never都很喜欢吃西瓜，而且一吃就是一堆的那种，zb立刻下定决心买了一堆西瓜。当他准备把西瓜送给C小加和never的时候，遇到了一个难题，never和C小加不在一块住，只能把西瓜分成两堆给他们，为了对每个人都公平，他想让两堆的重量之差最小。每个西瓜的重量已知，你能帮帮他么？

输入

多组测试数据 (≤ 1500)。数据以EOF结尾

第一行输入西瓜数量N ($1 \leq N \leq 20$)

第二行有N个数， $W_1, \dots, W_n$ ($1 \leq W_i \leq 100000$)分别代表每个西瓜的重量

输出

输出分成两堆后的质量差

样例输入

5 5 8 13 27 14

样例输出

3

```

1  #include <iostream>
2  #include <cstdio>
3  #include <string.h>
4  using namespace std;
5  int a[21], v, n, m;
6  void dfs(int i, int ans)
7  {
8      if (ans > v) return; // 超过一半退出
9      if (i > n)
10     { // 判断完毕
11         if (m < ans) m = ans;
12         return;
13     } // 对于每个西瓜两种选择, 取和不取
14     dfs(i+1, ans+a[i]);
15     dfs(i+1, ans);
16 }

```

```

17 int main()
18 {
19     int sum;
20     while (~scanf("%d", &n))
21     {
22         sum = 0;
23         for (int i=1; i<=n; i++)
24         {
25             scanf("%d", &a[i]);
26             sum += a[i];
27         }
28         v = sum/2; // 把一半体积作为边界
29         m = 0;
30         dfs(1, 0);
31         cout << sum - 2*m << endl;
32     }
33     return 0;
34 }

```



递归存在的问题

- 递归思想虽然很好理解，特别是对于一些可以用递推式子表示的问题。然而，**使用递归的代价**是十分巨大的：它会消耗大量的内存！！！！
- 函数调用时需要用到堆栈，而堆栈的资源是十分有限的。
- 在例8.3中，使用递归思想求Fibonacci数列的第n项。单独求**Fibonacci(20)**，递归调用Fibonacci()函数就达到**21891**次！！！！



今天的课程结束啦.....



下课了...
同学们再见!