哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

规格严格 功夫到家

第7讲 递归

教材7.4~7.5节

MOOC第7周

哈尔滨工业大学

苏小红

sxh@hit.edu.cn

你想改变世界吗？

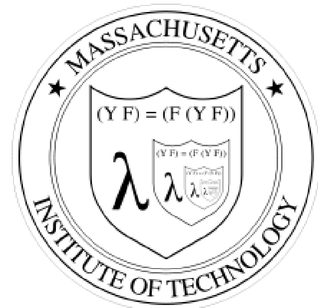
- 我梦想改变这个世界。
  * 当我成熟以后，我发现我不能够改变这个世界，我终将目标缩短了些，决定只想改变这个国家。
  * 当我进入暮年以后，我发现我不能够改变我的国家，我的最后愿望仅仅是改变一下我的家庭。但是这也不可能。

Westminster Abbey

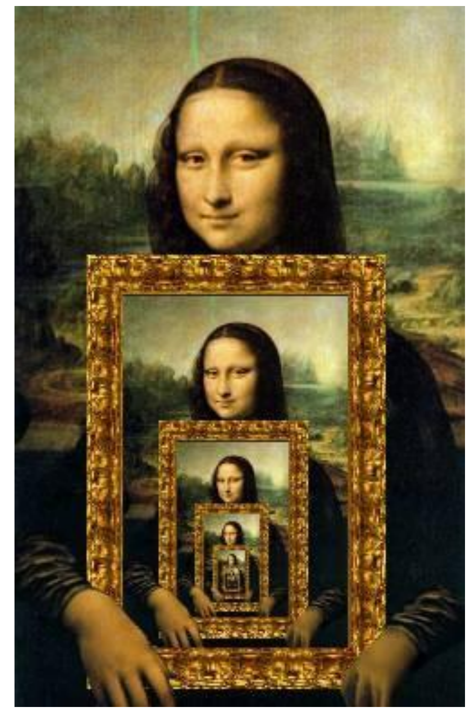
- 当我现在躺在床上，行将就木时，我突然意识到——
  * 如果一开始我仅仅去改变我自己，然后作为一个榜样，我可能改变我的家庭；在家人的帮助和鼓励下，我可能为国做一些事情。
  * 然后，谁知道呢？我甚至可能改变这个世界。

何为递归？

- **你站在桥上看风景，看风景的人在楼上看你。明月装饰了你的窗子，你装饰了别人的梦。——卞之琳《断章》**



电影《盗梦空间》——梦中梦



何为递归？

- **一种问题求解的策略**
  - **假设你已经走了999步，那么你再走一步即可迈出第1000步**
  - **假设你已经走了998步，那么你再走一步即可迈出第999步**
  - **……**
  - **直到你迈出第1步，递归结束**

递归的数学基础是什么？

生活中的递归实例

- 某家有5个兄弟，客人问老大的年龄，他说比老二大2岁；
- 问老二，他说比老三大2岁；
- 问老三，他说比老四大2岁；
- 问老四，他说比老五大2岁。
- 老五最乖，说自己10岁，问老大多大年纪。

递归函数（Recursive Function）

$$\text{age}(n)=\begin{cases}10 & (n=1)\\ \text{age}(n-1)+2 & (n>1)\end{cases}$$

```c
#include <stdio.h>
unsigned int CalAge(unsigned int n);
int main(void)
{
   unsigned int n = 5;
   printf("%d\n", CalAge(n));
   return 0;
}
```

```c
unsigned int CalAge(unsigned int n)
{
    if (n == 1)
    {
        return 10;
    }
    else
    {
        return CalAge(n-1) + 2;
    }
}
```

递归调用（Recursive Call）

递归函数的基本要素

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

```
unsigned int CalAge(unsigned int n)
{
    if (n == 1)
         return 10;
    else
         return CalAge(n-1) + 2;
}
```

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

$$\mathrm{age}(n) = \begin{cases} 10 & (n = 1) \\ \mathrm{age}(n-1) + 2 & (n > 1) \end{cases}$$

```
//条件递归
if (基本条件) //控制递归调用结束的条件，递归的出口
    return  递归公式的初值;
else              //一般条件定义了递归关系，控制递归调用向基本条件的方向转化
    return  递归函数调用的返回值;
```

递归函数是怎样调用的？

main()

```
Fact(3)
{
  if  (3==0 || 3==1)
     return 1;
  else
     return 3 * Fact(3-1);
}
```

```
Fact(2)
{
  if (2==0 || 2==1)
     return 1;
  else
     return 2 * Fact(2-1);
}
```

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

递推阶段（递归前进阶段）

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

回归阶段（递归返回阶段）

何时结束递归？

Fact(3) = 3 * Fact(2)　　递推阶段

Fact(2) = 2 * Fact(1)

Fact(1) = 1　　终止条件

Fact(2) = 2 * 1　　回归阶段

Fact(3) = 3 * 2

6

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

若无基本条件，或一般条件不能转化为基本条件？

递推阶段　递归前进

Fact (3) = 3 * Fact (2) = 3 * 2 = 6

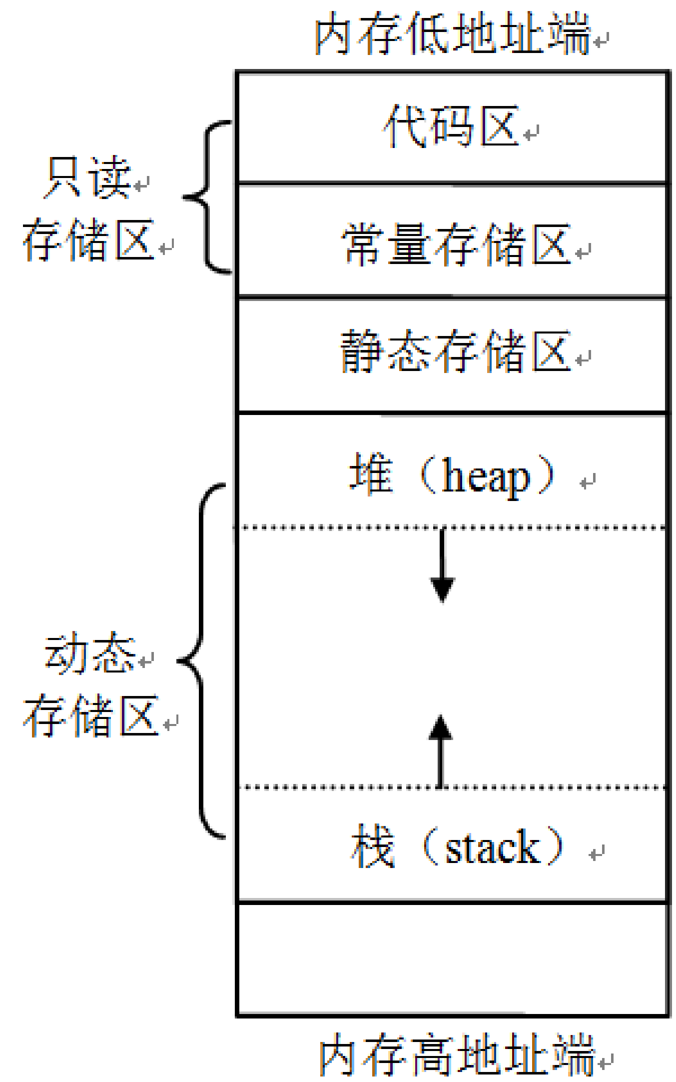
Fact (2) = 2 * Fact (1) = 2 * 1 = 2

Fact (1) = 1

递归返回　回归阶段

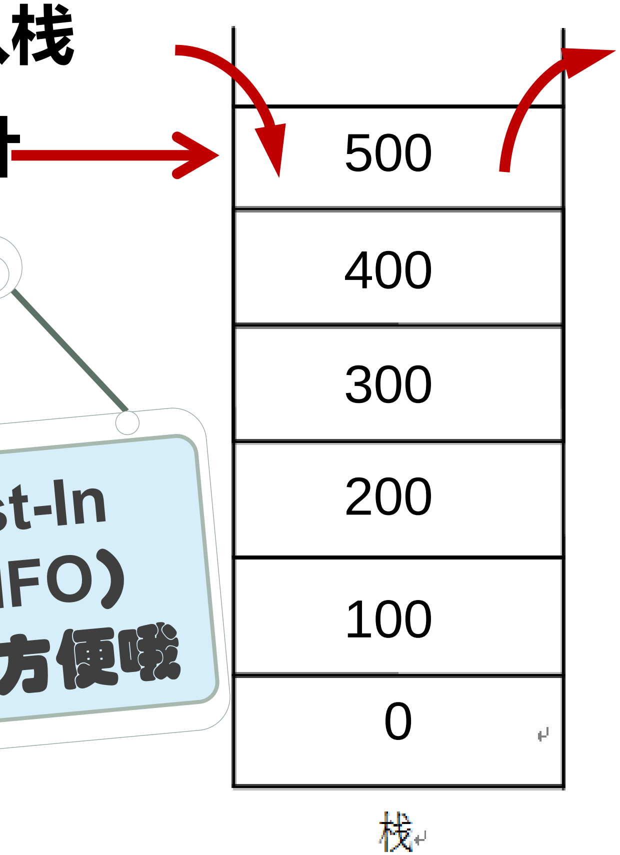
递归涉及的数据结构

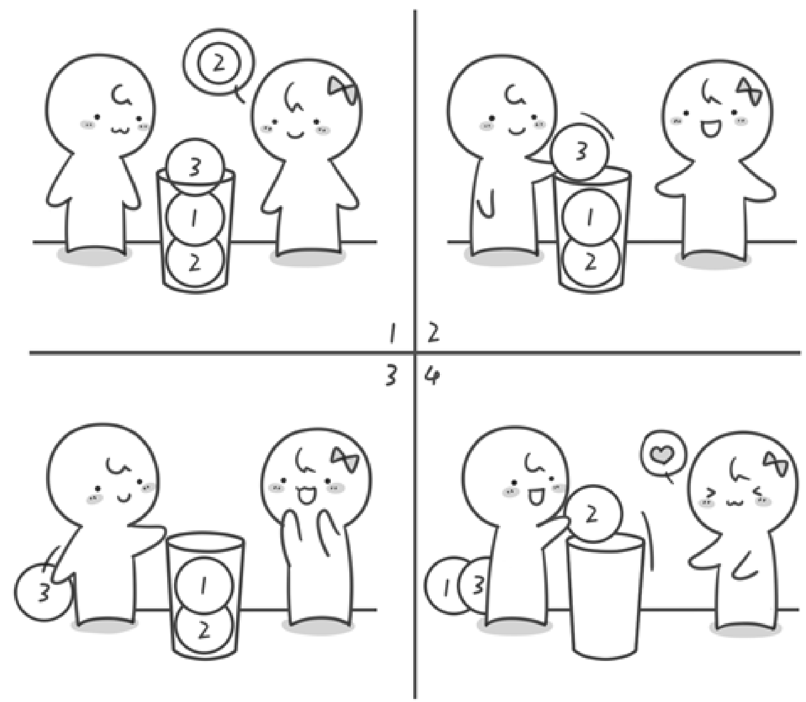
- 只读存储区
  - 存放机器代码和常量等只读数据
- 静态存储区
  - 存放程序中的全局变量和静态变量等
  - 静态——发生在程序编译或链接时
- 动态存储区
  - 包括堆和栈。用于保存与函数调用相关的信息的栈空间，称为函数调用栈
  - 动态——发生在程序载入和运行时

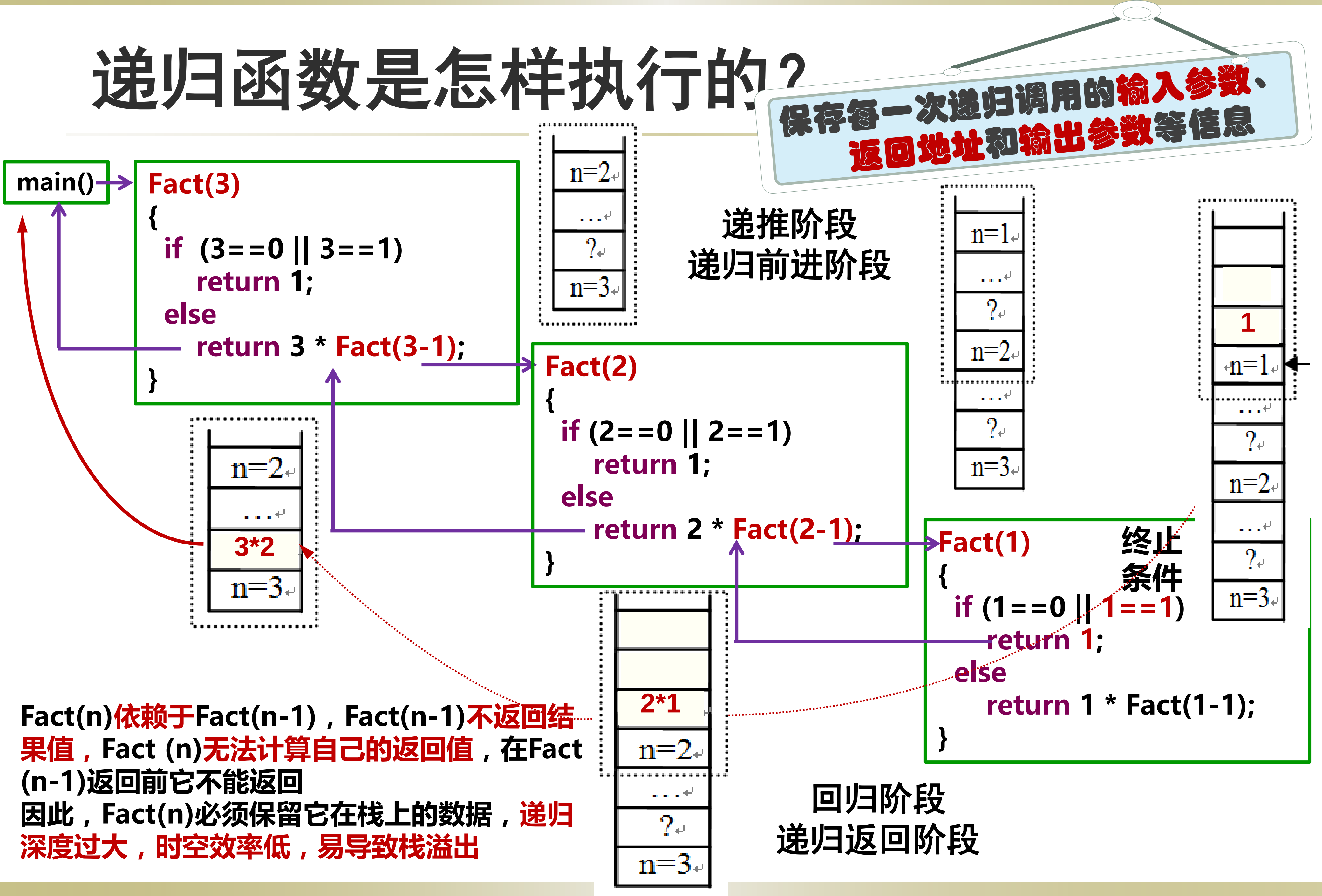


何为栈（stack）？

- **运算受限的线性表**——仅允许在表的一端插入和删除运算
- 栈溢出（Stack Overflow）



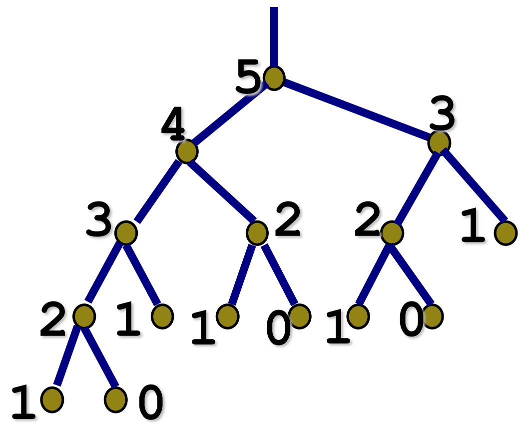




递归函数是怎样执行的？
保存每一次递归调用的输入参数、返回地址和输出参数等信息
main()
Fact(3)
{
if (3==0 || 3==1)
return 1;
else
return 3 * Fact(3-1);
}
Fact(2)
{
if (2==0 || 2==1)
return 1;
else
return 2 * Fact(2-1);
}
Fact(1)
{
if (1==0 || 1==1)
return 1;
else
return 1 * Fact(1-1);
}
终止条件
递推阶段
递归前进阶段
回归阶段
递归返回阶段
n=2
…
?
n=3
n=1
3*2
2*1
1
Fact(n)依赖于Fact(n-1)，Fact(n-1)不返回结果值，Fact (n)无法计算自己的返回值，在Fact (n-1)返回前它不能返回
因此，Fact(n)必须保留它在栈上的数据，递归深度过大，时空效率低，易导致栈溢出

递归 vs 迭代

计算Fibonacci数列**第n项**



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

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

计算Fib(5)需15次函数调用

- 优点：简洁、直观、精炼
- 缺点：
  - 重复计算多，递归层数过深时函数调用开销大，时空效率低，易导致栈溢出

- 建议：对时空效率要求高的场合，用迭代递推代替递归实现

```c
#include  <stdio.h>
int MonkeyEatPeach(int days);
int main()
{
   int x, days;
   printf("Input days:");
   scanf("%d", &days);
   x = MonkeyEatPeach(days);
   printf("x = %d\n", x);
   return 0;
}
```

```c
int MonkeyEatPeach(int days)
{
    int x = 1;
    while (days > 1)
    {
        x = 2 * (x + 1);
        days--;
    }
    return x;
}
```

```c
int MonkeyEatPeach(int days) //由第days天推出第days-1天
{
   if (days == 1)  //递归结束条件对应循环结束条件
       return 1;   //递归结束时的返回值对应递推初值
   else
       return 2 * (MonkeyEatPeach(days-1) + 1); //对应递推式
}
```

$x_n = 1 \qquad n = 10$

$x_n = 2 \times (x_{n+1} + 1) \qquad 1 \le n < 10$

- 原则上迭代程序都能转换为等价的递归程序，但反之不一定

设置

多选题

下列说法正确的是（）

A 基本条件是控制递归调用结束的条件，通常是递归公式的初值

B 一般条件定义了递归公式中的递推关系，控制递归调用向基本条件的方向转化

C n个汉诺塔的移动次数是利用等差数列求和公式计算出来的

D n个汉诺塔的移动次数是利用等比数列求和公式计算出来的

E 递归的主要问题是空间效率低，而不是时间效率低

提交

n位逆序数

- 从键盘任意输入一个整数n，编程将其逆序输出。
  - 如果n是负数，则不输出其负号。
  - 如果n逆序后的首位数字是0，则0也输出。
  - 思考：这个函数有什么局限性？
  - 思考：如何改为不输出首位数字0？

```c
int main()
{
  int n;
  printf("Input n:");
  scanf("%d", &n);
  Reverse(fabs(n));
  return 0;
}
```

```c
void Reverse(int n)
{
    if (n == 0) //对应循环结束条件
    {
        printf("\n");
    }
    else
    {
        printf("%d", n % 10);
        Reverse(n / 10); //对应递推式
    }
}
```

```c
void Reverse(int n)
{
    while (n != 0)
    {
        printf("%d", n % 10);
        n = n / 10;
    }
    printf("\n");
}
```

```c
void Reverse(int n)
{
    if (n == 0)
    {
        printf("\n");
    }
    else
    {
        printf("%d", n % 10);
        Reverse(n / 10);
    }
}
```

```c
void Reverse(int n)
{
    while (n != 0)
    {
        printf("%d", n % 10);
        n = n / 10;
    }
    printf("\n");
}
```

```c
int Reverse(int n) //被多次调用
{
    int a;
    //静态局部变量
    static int sum = 0;//仅第一次调用执行

    if (n == 0)
    {
        return sum;
    }
    else
    {
        a = n % 10;
        sum = sum * 10 + a;
        return Reverse(n / 10);
    }
}
```

```c
int Reverse(int n) //仅调用1次
{
    //动态局部变量
    int a, sum = 0;//每次调用都执行

    while (n != 0)
    {
        a = n % 10;
        sum = sum * 10 + a;
        n = n / 10;
    }
    return sum;
}
```

从n位逆序数到回文数

- 从键盘任意输入一个整数n，判断其是否是回文数。
  - 如果n是负数，则不输出其负号。

```c
//函数功能：返回n的逆序数
int Reverse(int n)
{
    int a, sum = 0;

    while (n != 0)
    {
        a = n % 10;
        sum = sum * 10 + a;
        n = n / 10;
    }
    return sum;
}
```

```c
int main()
{
  int n;
  printf("Input n:");
  scanf("%d", &n);
  if (IsPalindrome(fabs(n)))
      printf("Yes\n");
  else
      printf("No\n");
  return 0;
}
```

```c
//函数功能：判断n是否是回文数
int IsPalindrome(int n)
{
    int m;
    m = Reverse(n);
    return (m == n) ? 1 : 0;
}
```

思考题：回文数的形成

- 任取一个十进制整数，将其倒过来后与原来的整数相加，得到一个新的整数后，再将其倒过来后与原来的整数相加
- 以此类推，重复以上步骤，最终可得到一个回文数，请编程验证。

F:\c\test\bin\Debug\test.exe

```
Input n:1370
The generation process of palindrome:
[1]:1370+731=2101
[2]:2101+1012=3113
```

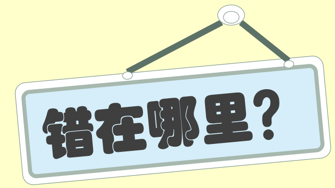
```c
int main()
{
    int n, m, i, flag;
    printf("Input n:");
    scanf("%d", &n);
    printf("The generation process of palindrome:\n");
    for (i=1, flag=0; !flag; i++)
    {
        m = Reverse(n);
        flag = IsPalindrome(n + m);
        printf("[%d]:%d+%d=%d\n", i, n, m, n + m);
        n = n + m;
    }
    return 0;
}
```

```c
int Reverse(int n) //非递归实现
{
    int a, sum = 0;

    while (n != 0)
    {
        a = n % 10;
        sum = sum * 10 + a;
        n = n / 10;
    }
    return sum;
}
```

```c
//函数功能：判断n是否是回文数
int IsPalindrome(int n)
{
    int m;
    m = Reverse(n);
    return (m == n) ? 1 : 0;
}
```

```c
int main()
{
    int n, m, i, flag;
    printf("Input n:");
    scanf("%d", &n);
    printf("The generation process of palindrome:\n");
    for (i=1, flag=0; !flag; i++)
    {
        m = Reverse(n);
        flag = IsPalindrome(n + m);
        printf("[%d]:%d+%d=%d\n", i, n, m, n + m);
        n = n + m;
    }
    return 0;
}
```




```c
int Reverse(int n) //递归实现 ??????
{
    int a;
    static int sum = 0;
    if (n == 0)
    {
        return sum;
    }
    else
    {
        a = n % 10;
        sum = sum * 10 + a;
        return Reverse(n / 10);
    }
}
```

```c
//函数功能：判断n是否是回文数
int IsPalindrome(int n)
{
    int m;
    m = Reverse(n);
    return (m == n) ? 1 : 0;
}
```

计算最大公约数

（1）穷举法

穷举范围： a和b的最大公约数不可能比t=min(a,b)大

从t开始逐次减1，即检验[t,1]间的所有整数

判定条件： 第一个满足公约数条件（同时被a和b整除）的t

```
int Gcd(int a, int b)
{
   int i, t;
   if (a <= 0 || b <= 0)      return -1;
   t = a < b ? a : b;
   for (i=t; i>0; i--)
   {
        if (a % i == 0 && b % i == 0)
            return i;
   }
   return 1;
}
```

计算最大公约数

（2）欧几里德算法（辗转相除法）

对正整数a和b，连续进行求余运算

r = a % b

直到余数r为0为止

此时非0的除数，即为所求

若r≠0，则b作为新的a，r作为新的b

即Gcd(a, b)=Gcd(b, r)

重复a % b运算，直到r=0时为止。

```
int Gcd(int a, int b)
{
    int r;
    if (a <= 0 || b <= 0)
    {
        return -1;
    }
    do{
        r = a % b;
        a = b;
        b = r;
    }while (r != 0);
    return  a;
}
```

计算最大公约数

（3）辗转相除法的递归实现

Gcd(a, b)=Gcd(b, r)

```c
int Gcd(int a, int b)
{
   int r;
   if (a <= 0 || b <= 0)
   {
        return -1;
   }
   do{
        r = a % b;
        a = b;
        b = r;
   }while (r != 0);
   return  a;
}
```

```c
int Gcd(int a, int b)
{
   if (a <= 0 || b <= 0)
   {
        return -1;
   }
   if (a%b == 0)
        return b;
   else
        return Gcd(b, a%b);
}
```

递归结束条件对应循环结束条件

大整数取余效率低?

计算最大公约数

（4）更相减损术（《九章算术》）——递归方法

性质1　如果a>b，则Gcd(a, b) = Gcd(a-b, b)
性质2　如果b>a，则Gcd(a, b) = Gcd(a, b-a)
性质3　如果a=b，则Gcd(a, b) = a = b

```
int Gcd(int a, int b)
{
    if (a <=0 || b <= 0) return -1;
    if (a == b)
        return a;
    else if (a > b)
        return Gcd(a-b, b);
    else
        return Gcd(a, b-a);
}
```

相当于辗转相减法，避免了取余，但算法不稳定

计算最大公约数

（5）更相减损术——迭代方法

性质1 如果a>b，则Gcd(a, b) = Gcd(a-b, b)
性质2 如果b>a，则Gcd(a, b) = Gcd(a, b-a)
性质3 如果a=b，则Gcd(a, b) = a = b

```
int Gcd(int a, int b)
{
    if (a <=0 || b <= 0)
    {
        return -1;
    }
    if (a == b)
        return a;
    else if (a > b)
        return Gcd(a-b, b);
    else
        return Gcd(a, b-a);
}
```

```
int Gcd(int a, int b)
{
    if (a <=0 || b <=0)
    {
        return -1;
    }
    while (a != b)
    {
        if (a > b)
        {
            a = a - b;
        }
        else if (b > a)
        {
            b = b - a;
        }
    }
    return a;
}
```

讨论

- 怎样知道这个两个函数的被调用次数？

```
int Gcd(int a, int b)

{
   if (a <= 0 || b <= 0)
         return -1;
   if (a%b == 0)
       return b;
   else
       return Gcd(b, a%b);
}
```

```
int Gcd(int a, int b)
{
     if (a <=0 || b <= 0)
         return -1;
     if (a == b)
          return a;
     else if (a > b)
          return Gcd(a-b, b);
     else
          return Gcd(a, b-a);
}
```

讨论

- **全局变量**：在所有函数之外定义的变量
  - 作用域：从定义位置开始到本程序结束

```c
int Gcd(int a, int b)
{
   count++;
   if (a <= 0 || b <= 0)
        return -1;
   if (a%b == 0)
       return b;
   else
       return Gcd(b, a%b);
}
```

```c
int Gcd(int a, int b)
{
    count++;
    if (a <=0 || b <= 0)
        return -1;
    if (a == b)
        return a;
    else if (a > b)
        return Gcd(a-b, b);
    else
        return Gcd(a, b-a);
}
```

观察函数调用栈（1101，1100），看谁的递归次数多？

全局变量的利与弊

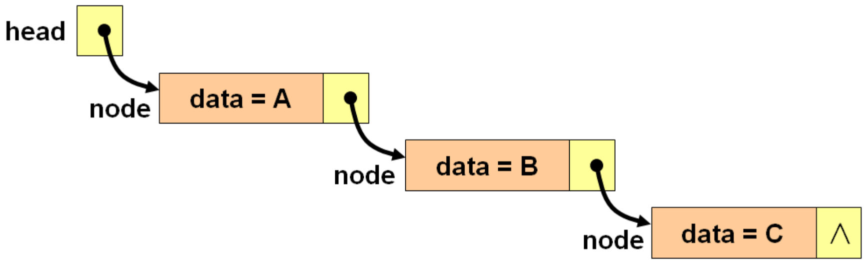
- 建议尽量不用
  - 破坏了函数的封装性，不能实现信息隐藏
  - 依赖全局变量的函数很难复用，维护也困难
- 何时可以用？
  - 当多个函数必须共享同一个固定类型的变量时
  - 当少数几个函数必须共享大量数据时
  - 例如飞机大战游戏中的飞机、子弹、敌机位置和画布尺寸

课后思考题

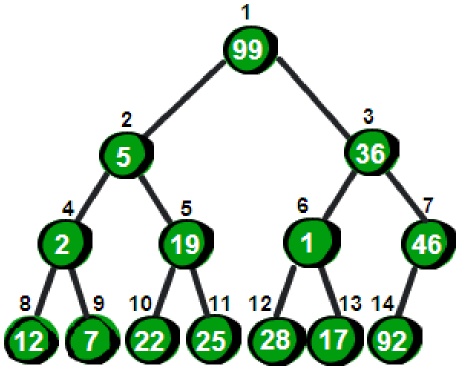
- 如何计算最小公倍数？

什么情况下考虑使用递归？

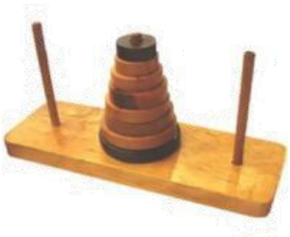
- **数学定义**是递归的



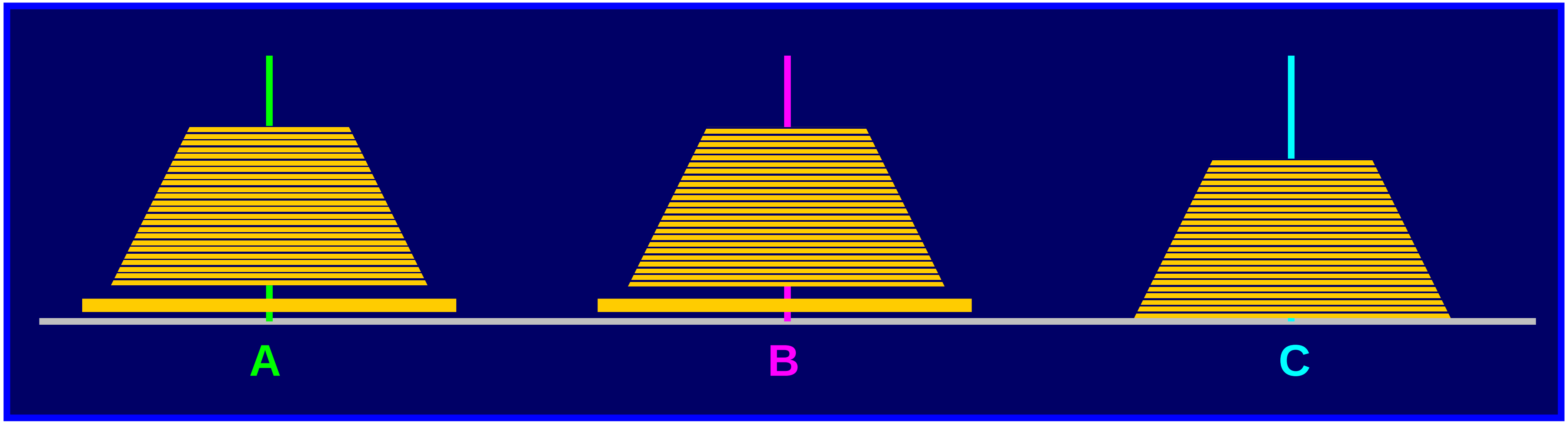
- **数据结构**是递归的
  - 如队列、链表、树和图
- **问题的解法**是递归的
  - 如Hanoi塔，骑士游历、八皇后问题（回溯法）



- 汉诺塔（Hanoi）问题
  - 印度神话，上帝创造世界时作了三根金刚石柱子，第一根上从下往上按大小顺序摞着64片黄金圆盘，上帝命令婆罗门把圆盘从下开始按大小顺序重新摆放到第二根上，规定每次只能移动一个圆盘，在小圆盘上不能放大圆盘



递归求解汉诺塔的数学基础是什么？



- 数学归纳法
  - 假设n-1个圆盘的汉诺塔问题已解决
  - 将“上面n-1个圆盘”看成一个整体

移动n个圆盘 ➡ 移动n−1个圆盘

✓ 将“上面n-1个圆盘”从A移到C

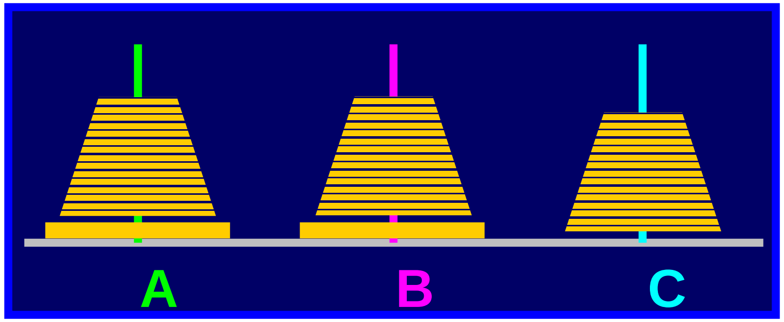
✓ 将第n号圆盘从A移到B

✓ 将“上面n-1个圆盘”从C移到B

汉诺塔问题的递归函数实现

- ✓ 将“n个圆盘”借助于C从A移到B

```
void Hanoi(int n,char a,char b,char c)
{
  if (n == 1)
  {
      Move(n, a, b);
  }
  else
  {
      Hanoi(n-1, a, c, b);
      Move(n, a, b);
      Hanoi(n-1, c, b, a);
  }
}
```



- ✓ 将“n-1个圆盘”从A移到C
- ✓ 将第n号圆盘从A移到B
- ✓ 将“n-1个圆盘”从C移到B

汉诺塔问题的移动次数？

* 当n=64时，需移动
  * 18446744073709551615，即1844亿亿次
  * 若按每次耗时1微秒计算，则64个圆盘的移动需60万年
* 谁知道这个数是怎样算出来的？

```
void Hanoi(int n,char a,char b,char c)
{
  if (n == 1)
  {
      Move(n, a, b);
  }
  else
  {
      Hanoi(n-1, a, c, b);
      Move(n, a, b);
      Hanoi(n-1, c, b, a);
  }
}
```

$$T(n) = 2T(n-1) + 1$$
$$2T(n-1) = 4T(n-2) + 2$$
$$4T(n-2) = 8T(n-3) + 4$$
$$\ldots\ldots$$
$$2^{n-2}T(2) = 2^{n-1}T(1) + 2^{n-2}$$

$$T(n) = 2^n-1$$

可以递归编程计算吗？

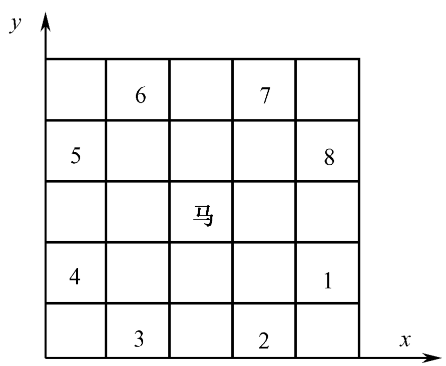
等比数列求和公式

$$S_n = n \times a_1 \quad (q = 1)$$
$$S_n = a_1 \cdot \frac{1-q^n}{1-q} = \frac{a_1 - a_n \cdot q}{1-q} \quad (q \neq 1)$$

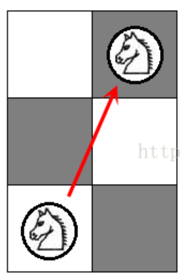
```
int HanoiTimes(int n)
{
  if (n == 1)
  {
      return 1;
  }
  else
  {
      return 2 * HanoiTimes(n-1) + 1;
  }
}
```

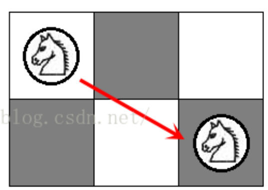
经典的递归问题

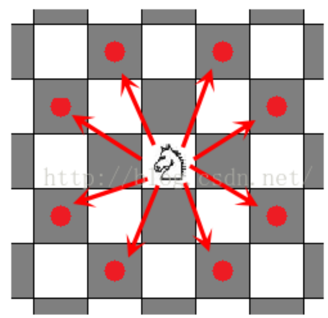
- **骑士游历问题**
  - 一块有N^2个格子的$N \times N$棋盘，一位骑士从初始位置(x0, y0)开始，按照“马跳日”规则在棋盘上移动。
  - 问：能否在N^2-1步内遍历棋盘上的所有位置，即每个格子刚好游历一次。如果能，请找出这样的游历方案来。
  - 用回溯法求解，用递归函数来实现

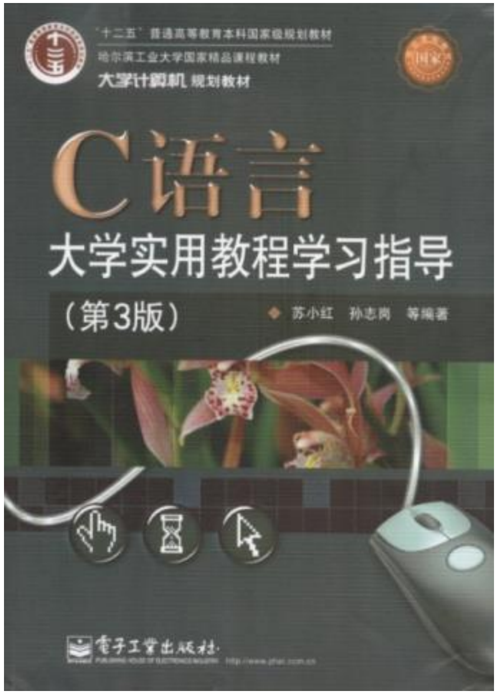


1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23



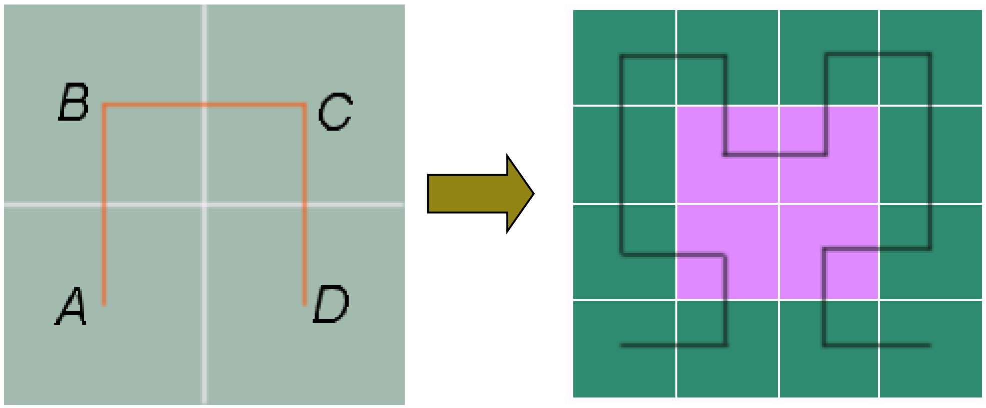


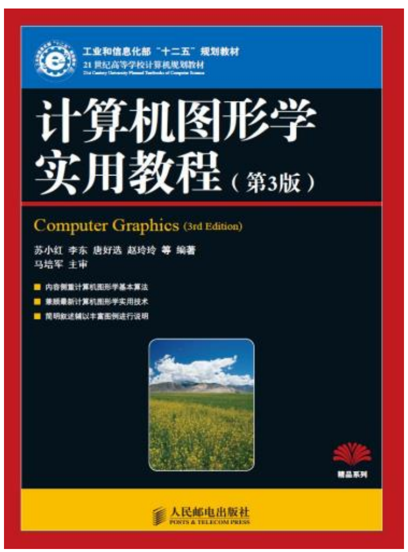


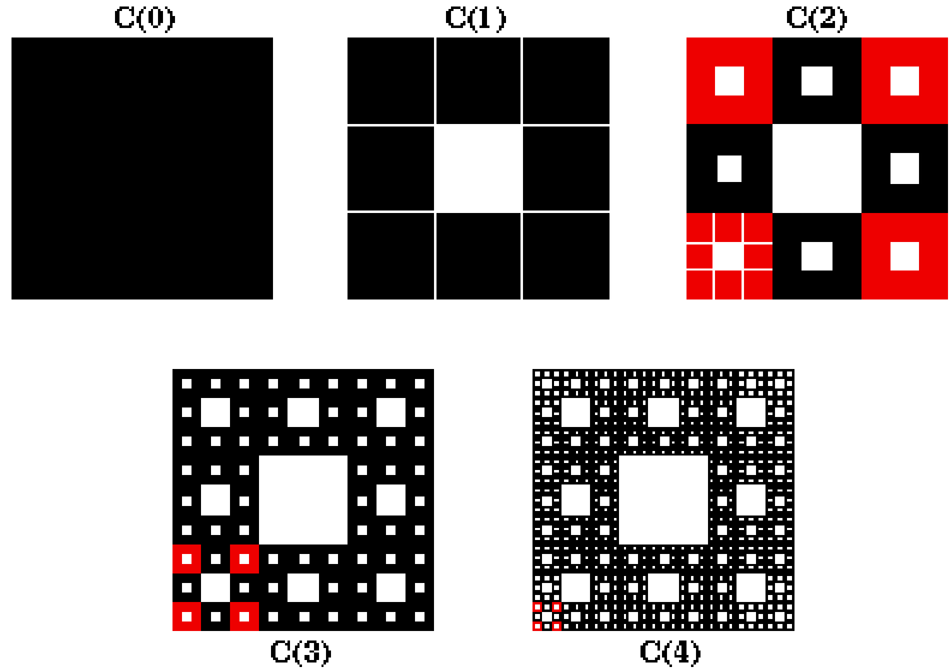


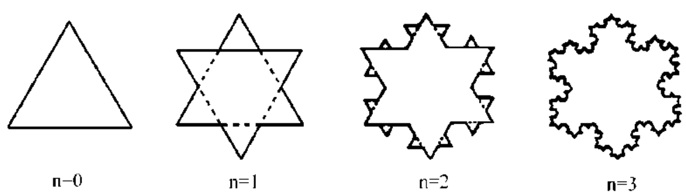
经典的递归问题

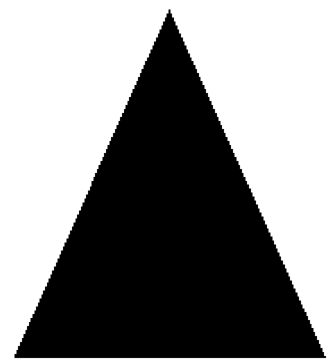
- **分形曲线**











挑战速度：计算n（<1000000）以内的所有完数

```
int IsPerfect(int x)
{
    int i, sum = 0;
    for (i=1; i<=x/2; i++)
    {
        if (x%i == 0)
        {
            sum = sum + i;
        }
    }
    return sum==x ? 1 : 0;
}
```

```
int IsPerfect(int x)
{
    int i;
    int sum = 1;
    int k = (int)sqrt(x);
    for (i=2; i<=k; i++)
    {
        if (x%i == 0)
        {
            sum += i;
            sum += x/i;
        }
    }
    return sum==x ? 1 : 0;
}
```

- 求解算法
  - i从1试到x/2，看i是否是x的真因子
  - 若x能被i整除，则累加到sum
  - 累加结束判断x是否等于sum，返回判断结果