我们国家第一艘航空母舰：辽宁舰

材料技术

航母动力技术

舰载机

舰机适配技术

新概念武器技术

……

辽宁舰

Divide-and-Conquer 分而治之

手机的设计

- 工业设计
  - 外观
  - 材质
  - UI
  - … …
- 硬件设计
- 结构设计
- 软件设计
  - Apple iOS
  - 应用软件
- 资源开发、质量测试…

Divide-and-Conquer 分而治之

函数调用

- 函数的调用是可以嵌套的。

ln(1+x)=x-x^2/2+x^3/3-……+(-1)^(k-1)*(x^k)/k
(|x|<1)

```
int main()
{
    compute_ln(x);
}
double compute_ln(double x)
{
    …… ……
    pow(x, k);
    …… ……
}
```

再分析这个例子

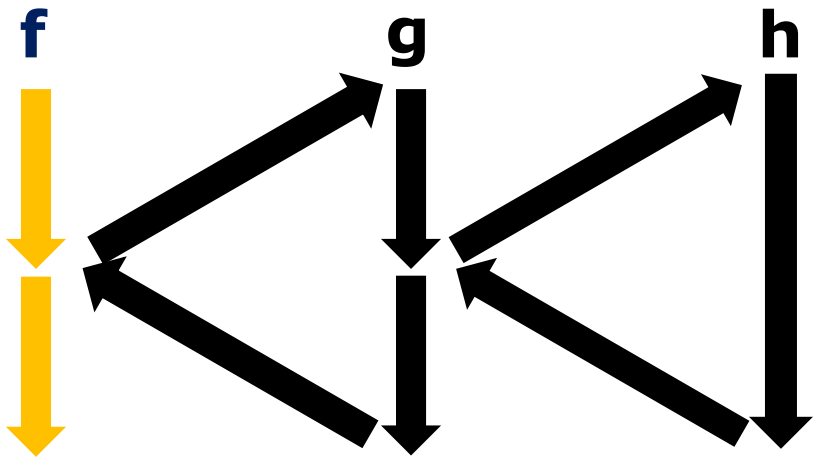
存在冗余计算吗？

有更优化的计算方法吗？

函数调用

- 函数的调用是可以嵌套的。

```
void h()
{ ......
}
void g()
{ ......
   h();
   ......
}
void f()
{ ......
   g();
   .......
}
```



函数的嵌套调用及返回

存在情况：函数在其函数体中直接或间接地调用了自己！

4.4 递归函数

郭 延 文

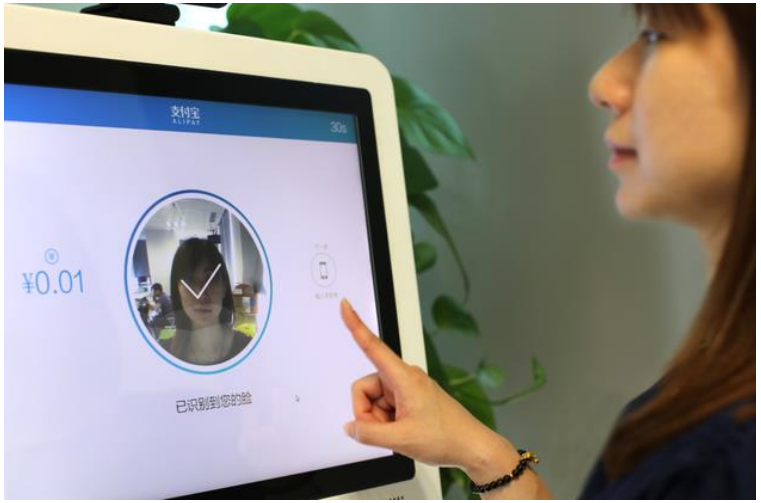
2019级计算机科学与技术系

IPhone’ s FaceID

专用人工智能的成功应用：人脸检测识别及应用

仅靠图像安全吗？活体检测！

认证

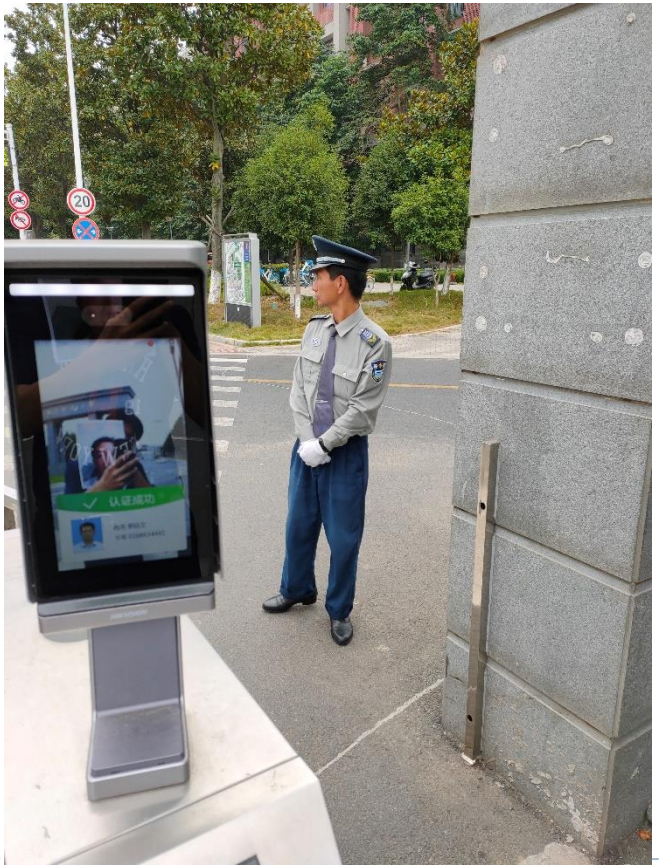


支付

视频分析及行业应用

公安应用

人脸检测识别 - 活体检测



活体检测

目标：判断是真人

方法：

- 无感：

颜色统计、纹理等外观特征

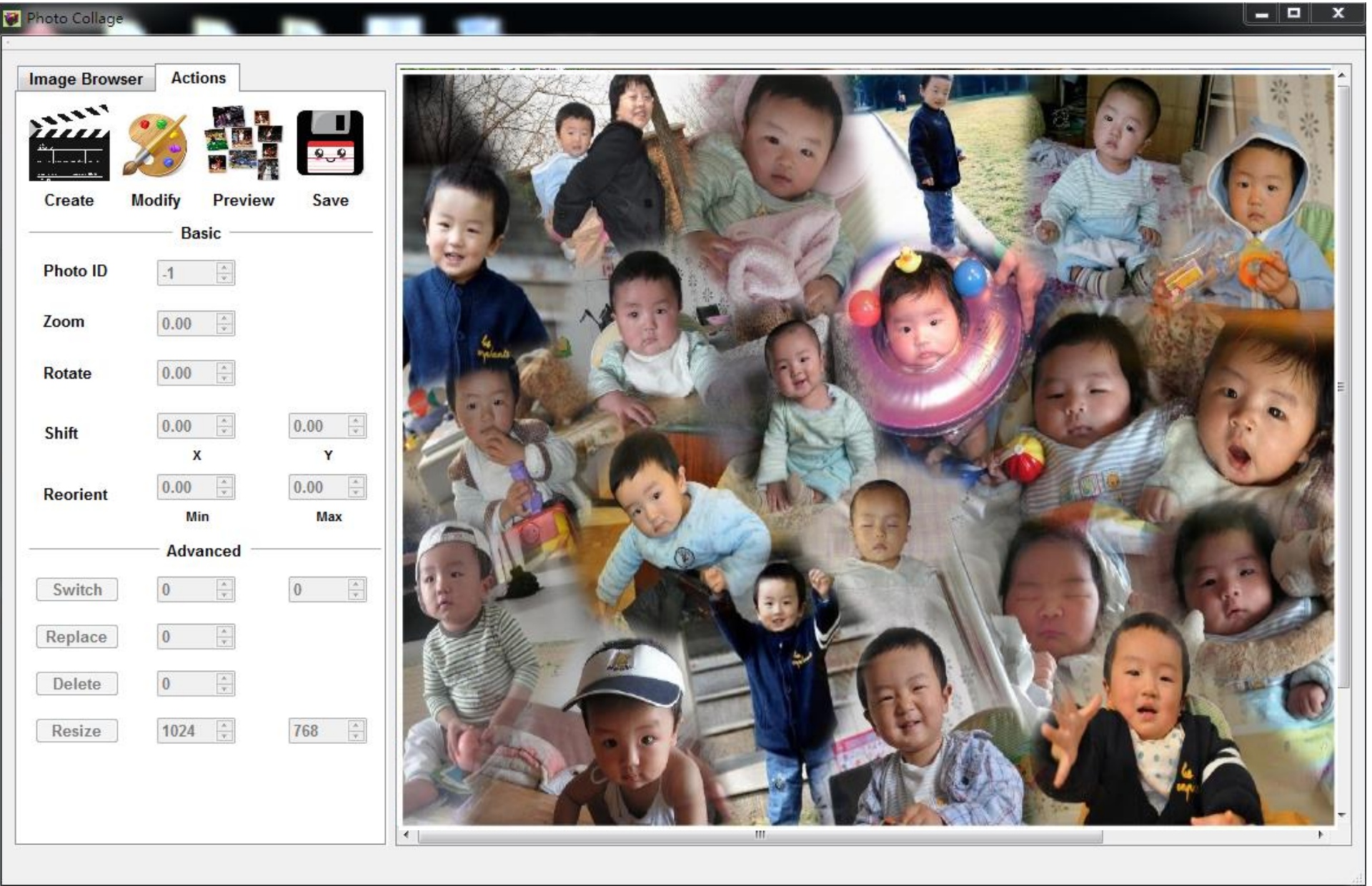
（红外）反射特性

微表情、动作

- 有感：

摇头、眨眼睛...

人脸检测的应用：照片Collage[本组研究成果]



专用人工智能的成功应用：
人脸检测识别在智慧教育中的应用[本组研究成果]

检测21人，实际22人

检测124人，实际127人

文文老师
正常上课

4.4 递归函数

郭 延 文

2019级计算机科学与技术系

- 如果一个函数在其函数体中直接或间接地调用了自己，则该函数称为**递归函数**。

- 直接递归

```
void f()
{   .......
    ... f() ...
    .......
}
```

- 间接递归

```
extern void g();
void f()
{   .......
    ... g() ...
    .......
}
void g()
{   ......
    ... f() ...
    ......
}
```

递归函数的作用

- 在程序设计中经常需要实现重复性的操作（一种实现重复操作的途径：循环）

- 实现重复操作的另一个途径是采用递归函数
- “分而治之”（Divide and Conquer）设计方法：
  - 把一个问题分解成若干个子问题，而每个子问题的性质与原问题相同，只是在规模上比原问题要小。每个子问题的求解过程可以采用与原问题相同的方式来进行。

- 递归函数为上述设计方法提供了一种自然、简洁的实现机制

例：求第n个fibonacci 数（递归解法）

兔生兔问题 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …… …

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

Imagine：你是一家工厂的老板，计算fibonacci数列的值来赚钱（卖给土著），现在有客户要求计算f(5)的值。但你这工厂只有计算f(1)和f(2)的机器，现要求f(5)，怎么办呢？

总经理一看计划表，原来求f(5)先求f(4)和f(3)就可以，于是就吩咐两个经理，一个求f(4)，另一个求f(3).

1. 求f(3)的经理窃喜，虽然他不能直接求出f(3),但f(1)和f(2)是可求的，于是他又派两手下计算f(1)和f(2)，结果很快出来，两报告说都是1，他一汇总求和得到2，就去找总经理报告了.

2. 求f(4)的哥们就比较倒霉，他一看，要求f(4)，就要求f(3)和f(2)，这f(2)好求啊，f(3)直接求不出，还要求f(2)和f(1)，麻烦。不管了,我是经理，这种小事还要我算，派两主管，一个求f(3)，另一个求f(2).手下求f(2)的人返回结果1，求f(3)的人,心想这么复杂的东西，还要我堂堂主管出马，再派两人（假设是职员）去求f(2)和f(1)，职员返回两个1给主管，汇总之后得到2，他报告给求f(4)的经理，经理根据两个主管的反馈结果得到1+2=3，他把结果返回给总经理。

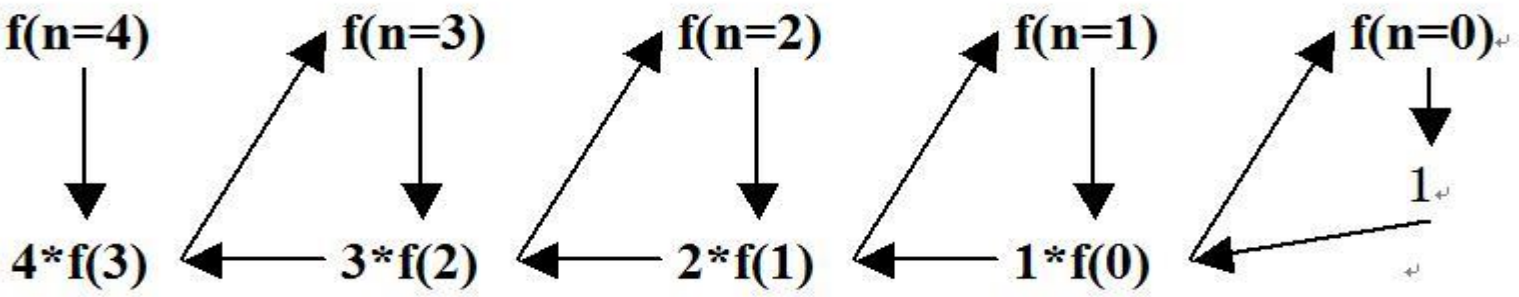
总经理根据两经理反馈的结果求和得到f(5)=2+3=5,于是他向你交差，说f(5)=5.

递归的执行：层层下派，层层返回
汇总后得到结果！

递归函数的执行过程

```
//用递归函数求n!
int f(int n)
{   if (n == 0)
        return 1;
    else
        return n*f(n-1);
}
```

当计算 f(4)时，可以按照下面的函数嵌套调用来理解：



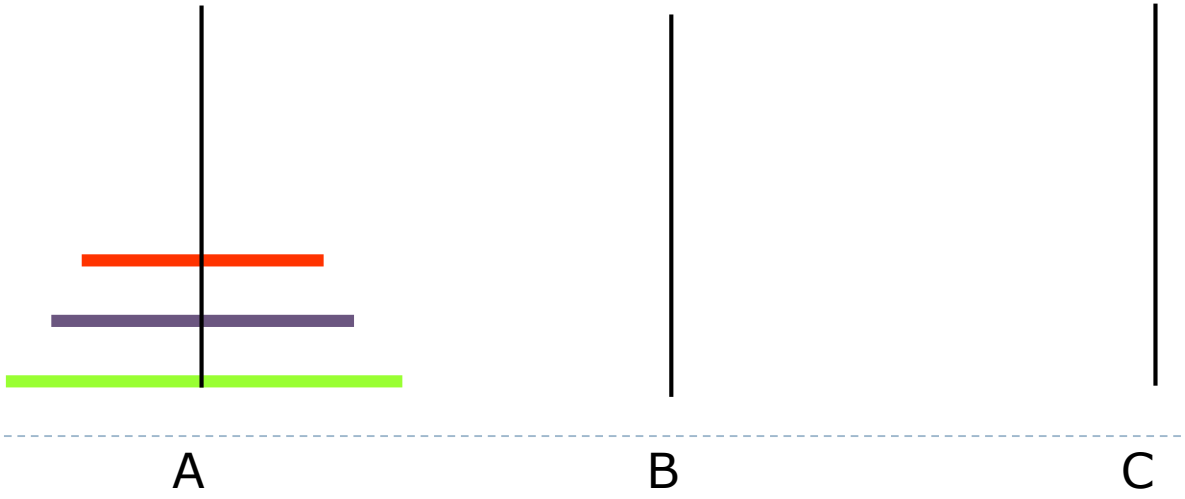
递归条件和结束条件

- 在定义递归函数时，一定要对两种情况给出描述：
  - 递归条件：指出何时进行递归调用，它描述了问题求解的一般情况，包括：分解和综合过程。
  - 结束条件：指出何时不需递归调用，它描述了问题求解的特殊情况或基本情况。

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

例：解汉诺塔问题

- **汉诺塔问题**：有A，B，C三个柱子，柱子A上穿有n个大小不同的圆盘，大盘在下，小盘在上。现要把柱子A上的所有圆盘移到柱子B上，要求每次只能移动一个圆盘，且大盘不能放在小盘上，移动时可借助柱子C。编写一个C++函数给出移动步骤，如：n=3时，移动步骤为：1:A→B, 2:A→C, 1:B→C, 3:A→B, 1:C→A, 2:C→B, 1:A→B。



- **当n=1：** 只要把1个圆盘从A移至B就可以了
  - cout << "1:A→B" << endl;
- **当n>1：** 我们可以把该问题分解成下面的三个子问题：
  1. 把n-1个圆盘从柱子A移到柱子C。
  2. 把第n个圆盘从柱子A移到柱子B。
     1. cout << n << "：A->B" << endl;
  3. 把n-1个圆盘从柱子C移到柱子B。

- **上面的子问题1和3与原问题相同，只是盘子的个数少了一个以及移动的位置不同；子问题2是移动一个盘子的简单问题。**

```
#include <iostream>
using namespace std;
void hanoi(char x,char y,char z,int n) //把n个圆盘从x表示的
                                       //柱子移至y所表示的柱子。
{  if (n == 1)
        cout << "1: " << x << "→" << y << endl; //把第1个
                        //盘子从x表示的柱子移至y所表示的柱子。
   else
   {    hanoi(x,z,y,n-1); //把n-1个圆盘从x表示的柱子移至
                          //z所表示的柱子。
        cout << n << ": " << x << "→" << y << endl;
              //把第n个圆盘从x表示的柱子移至y所表示的柱子。
        hanoi(z,y,x,n-1); //把n-1个圆盘从z表示的柱子移至
                          //y所表示的柱子。
   }
}
```

例：五人年龄问题： 五个人坐在一起问年龄，问第五个人几岁？他说比第四个人大2岁；第四个人说他比第三个人大2岁；第三个人说他比第二个人大2岁；第二个人说他比第一个人大2岁；第1个人说他自己10岁；

请问第5个人多大？

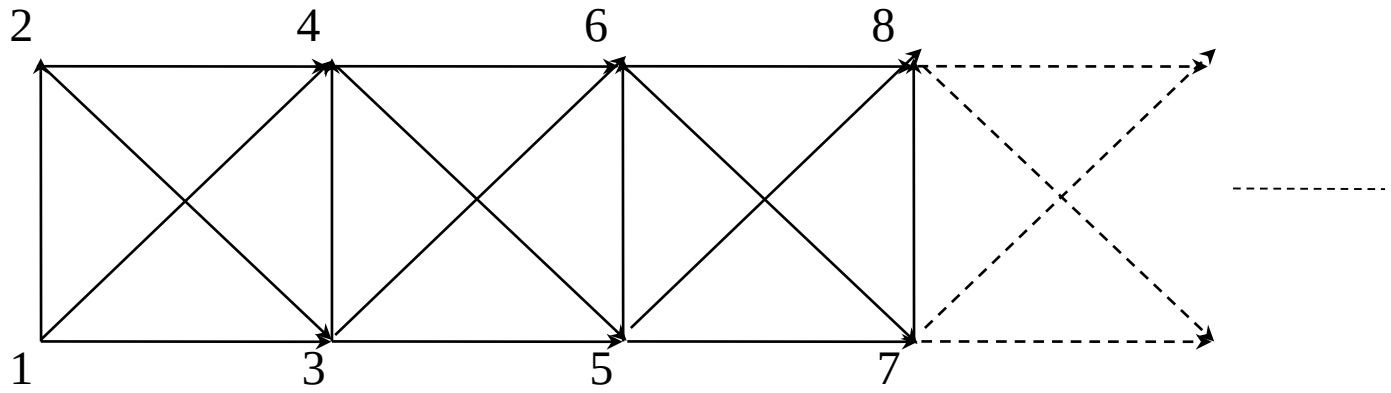
```
int age(int n)
{
    int x;
    if(n==1)
       x=10;
    else
       x=age(n-1)+2;
    return x;
}

void main()
{
    printf(“He is %d years old”, age(5));
}
```

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

该函数通过了递归（函数）实现，其实可以通过循环实现！

例：路径问题

- 根据图写一个递归函数：int Path(int n); 计算从结点1到结点n（n大于1）共有多少条不同的路径。



- 分析：从图可以看出，
- n为大于1的奇数时，可经过前2个结点到达，
- n为大于2的偶数时，可经过前3个结点到达。
- 所以，要想求从结点1到结点n的路径，必须先依次计算前面从结点1到结点n-1、n-2、n-3的路径数。
- 可以采用递归函数实现这个问题。

代码怎么写？

例：小牛问题：若一头小母牛，从出生起第四个年头开始每年生一头母牛（每头新生小牛也符合此规律），按此规律，第n年有多少头母牛？

year	未成熟母牛头数	成熟母牛头数	母牛总头数
1	1	0	1
2	1	0	1
3	1	0	1
4	+1	1	2
5	+1+1	1	3
6	+1+1+1	1	4
7	+1+1+1+1	1+1	6
8	+1+1+1+1+1+1	1+1+1	9

Fn = Fn-1 + Fn-3.

类似问题的规律：

Fn = Fn-1 + Fn-(m-1)

（n>m））

有时候动手“演算”可能有效，能找到规律！

代码怎么写？

```
int myGetCowR(int year)
{
     if ( year < 4 )
          return 1;
     else
          return myGetCowR(year-1) + myGetCowR(year-3);
}
```

```
    myGetCowR(int year, int m)   // 一般的情况
{
          if ( year < m )
                return 1;
          else
                return myGetCowR(year-1) + myGetCowR(year-m+1);
}
```

递归与循环的选择

- 对于一些递归定义的问题，用递归函数来解决会显得比较自然和简洁，而用循环来解决这样的问题，有时会很复杂，不易设计和理解。
- 在实现数据的操作上，它们有一点不同：
  - 循环是在同一组变量上进行重复操作（循环常常又称为迭代）
  - 递归则是在不同的变量组（属于递归函数的不同实例）上进行重复操作。
- 递归的缺陷：
  - 由于递归表达的重复操作是通过函数调用来实现的，而函数调用是需要开销的；
  - 栈空间的大小也会限制递归的深度。
  - 递归算法有时会出现重复计算。

有时候通过递归实现的函数可以通过循环实现！

内容回顾

- 递归函数
  - 一个函数在其函数体中直接或间接地调用了自己
  - 直接和间接递归
  - 递归和结束条件
  - 递归和循环

Q & A

程序中的变量

内容回顾

```
……
全局变量 //一般能被所有函数使用
……
函数（局部变量）// 复合语句
{
    局部变量

}
……
函数（局部变量）// 复合语句
{
    局部变量


}
……
```

内容回顾

变量的生存期（存储分配）

- 把程序运行时一个变量占有内存空间的时间段称为该变量的生存期。
  - 静态：从程序开始执行时就进行内存空间分配，直到程序结束才收回它们的空间；全局变量具有静态生存期
    - 具有静态生存期的变量，如果没有显式初始化，系统将把它们初始化成0
  - 自动：内存空间在程序执行到定义它们的复合语句（包括函数体）时才分配，当定义它们的复合语句执行结束时，它们的空间将被收回。局部变量和函数的参数一般具有自动生存期
  - 动态：内存空间在程序中显式地用new(c++)操作或malloc(c)库函数分配、用delete(c++)操作或free(c)库函数收回。动态变量具有动态生存期

内容回顾

C++标识符的作用域

- C++把标识符的作用域分成若干类，其中包括：
  - 局部作用域
  - 全局作用域
  - 文件作用域
  - 函数作用域
  - 函数原型作用域
  - 类作用域
  - 名空间作用域

生存期 V.S. 作用域

属性

生存期

变量占有内存空间的时间段

“时间”上的概念

作用域

标识符的有效范围

从“代码空间（篇幅）”上讲

```
…… 
int main()
{
    int i, num=0;
    int temp;
    for (i=1; i<11; i++)
    {
        // int temp;
        temp = i*i;
        num += temp;
    }
}
…
```

内容回顾

```
//模块1
namespace A
{  int x=1;
   void f()
   { ……
   }
}
```

```
//模块2
namespace B
{   int x=0;
    void f()
    { ……
    }
}
```

//模块3

1、

```
… A::x …  //A中的x
A::f();  //A中的f
… B::x …  //B中的x
B::f();  //B中的f
```

2、

```
using namespace A;
… x …  //A中的x
f();  //A中的f
… B::x …  //B中的x
B::f();  //B中的f
```

3、

```
using A::f;
… A::x …//A中的x
f();//A中的f
… B::x …  //B中的x
B::f();  //B中的f
```