



《计算概论A》课程 程序设计部分

函数的递归

李 戈

北京大学 信息科学技术学院 软件研究所

lige@sei.pku.edu.cn





本节内容

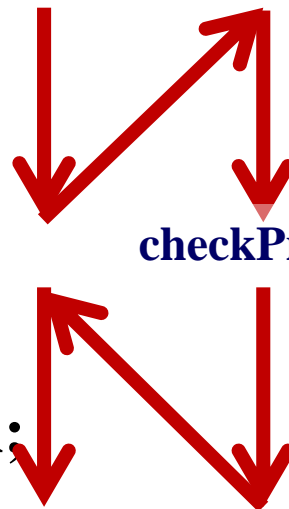
- 什么是递归
- 深入理解递归的过程
- 递归的作用
 - ◆ 用递归来完成递推
 - ◆ 模拟连续发生的动作
 - ◆ 进行“自动的分析”

函数的嵌套调用

```
#include <iostream>
#include <cmath>
using namespace std;
bool checkPrime(int);
int main()
{
    int a;
    cout << "请输入一个整数" << endl;
    while (cin >> a)
    {
        if (checkPrime(a))
            cout << "是质数" << endl;
        else
            cout << "不是质数" << endl;
    }
    return 0;
}
```

main()

checkPrime(a)



函数的嵌套调用

```
bool checkPrime(int number)
```

```
{
```

```
    int i, k;
```

```
    k = sqrt(number);
```

```
    for (i = 2; i <= k; i++)
```

```
    {
```

```
        if (number % i == 0)
```

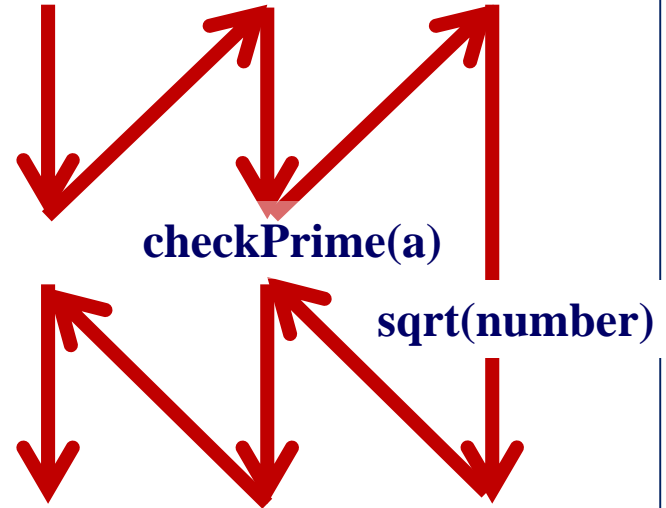
```
            return 0;
```

```
    }
```

```
    return 1;
```

```
}
```

main()



//只要有一个数被除尽

//则不是素数

//走到这一步，说明没能被除尽

思路整理

■ 我们已经知道

◆ 函数不能嵌套定义

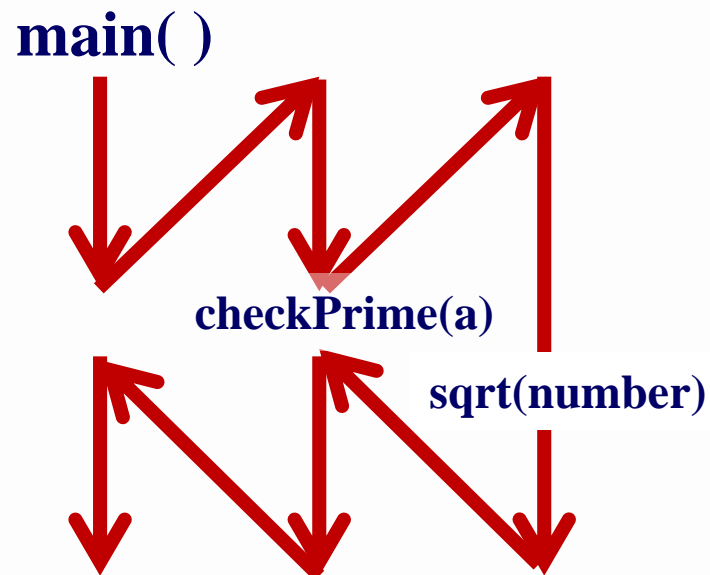
- 所有函数一律平等

◆ 函数可以嵌套调用

- 无论嵌套多少层，原理都一样

■ 问个问题

◆ 一个函数能调用“它自己”吗？



求 $n!$

```
#include<iostream>
using namespace std;
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n*fact(n-1);
}
int main() {
    cout << fact(4) << endl;
    return 0;
}
```

■ 问题：已知 n ，求 $n!$

$$n! = (n-1)! * n$$

$$(n-1)! = (n-2)! * (n-1)$$

.....

$$3! = 2! * 3;$$

$$2! = 1! * 2$$

$$1! = 1;$$

■ 可知

◆ $\text{fact}(n)$ 的值等于 $\text{fact}(n-1) * n$;

◆ $\text{fact}(1) = 1$;

思考一下

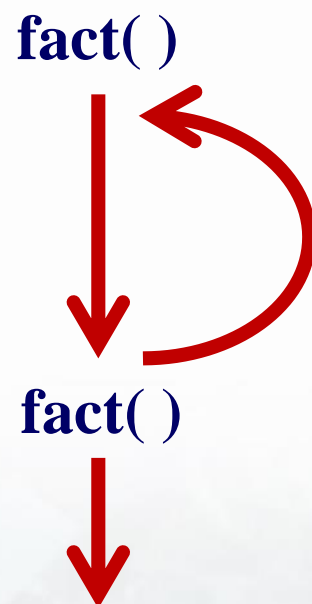
```
#include<iostream>
using namespace std;
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n*fact(n-1);
}
int main() {
    cout << fact(4) << endl;
    return 0;
}
```

- 这个称为 “递归调用”

- 什么是递归？

他说

- ◆ 一个函数在其定义中直接或间接调用自身的一种方法；



函数的执行

```
#include <iostream>
using namespace std;
float max(float a, float b)
{
    (1) 初始化max();
    (2) 传递参数;
    (3) 保存当前现场;
    return b;
}
```

```
int main()
{
    (1) 接收函数的返回值;
    (2) 恢复现场, 从断点处继续执行;
    return 0;
}
```

main()

```
int main()
{
    int m = 3, n = 4;
    float result = 0;
    result = max(m, n);
    cout << result;
    return 0;
}
```

max()

```
float max(float a, float b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

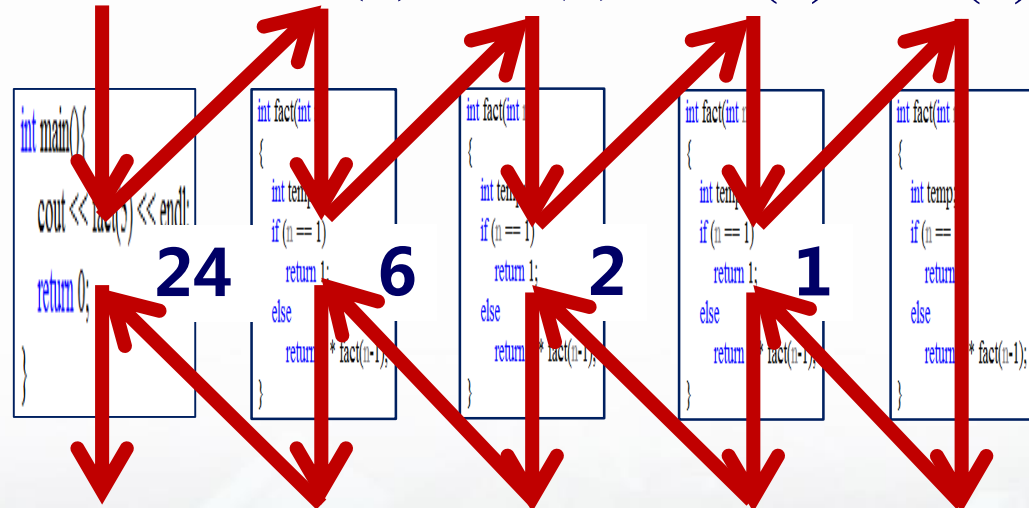

思考一下

```
#include<iostream>
using namespace std;
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n*fact(n-1);
}
int main() {
    cout << fact(4) << endl;
    return 0;
}
```

■ 我说

**递归调用与普通调用
没有区别！！！！**

main() fact(4) fact(3) fact(2) fact(1)





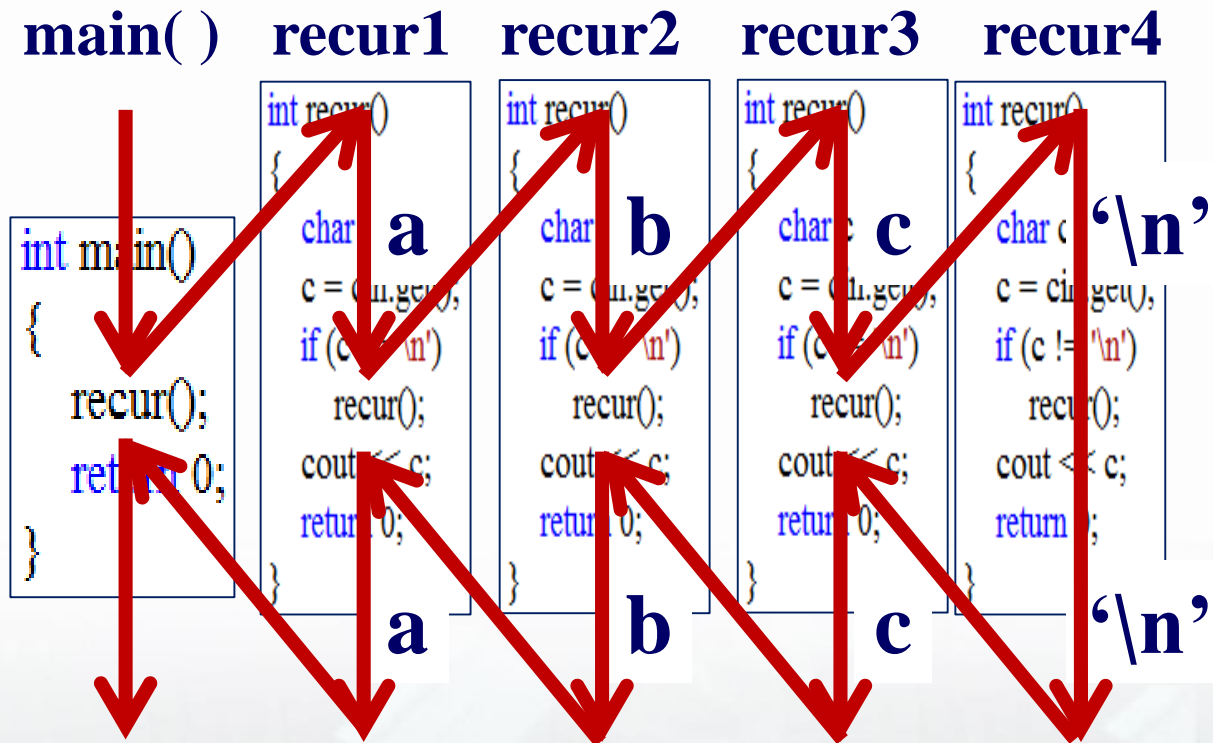
本节内容

- 什么是递归
- 深入理解递归的过程
- 递归的作用
 - ◆ 用递归来完成递推
 - ◆ 模拟连续发生的动作
 - ◆ 进行“自动的分析”

深入理解递归的过程

```
#include<iostream>
using namespace std;
int recur()
{
    char c;
    c = cin.get();
    if (c != '\n')
        recur();
    cout << c;
    return 0;
}
int main()
{
    recur();
    return 0;
}
```

abc
cba

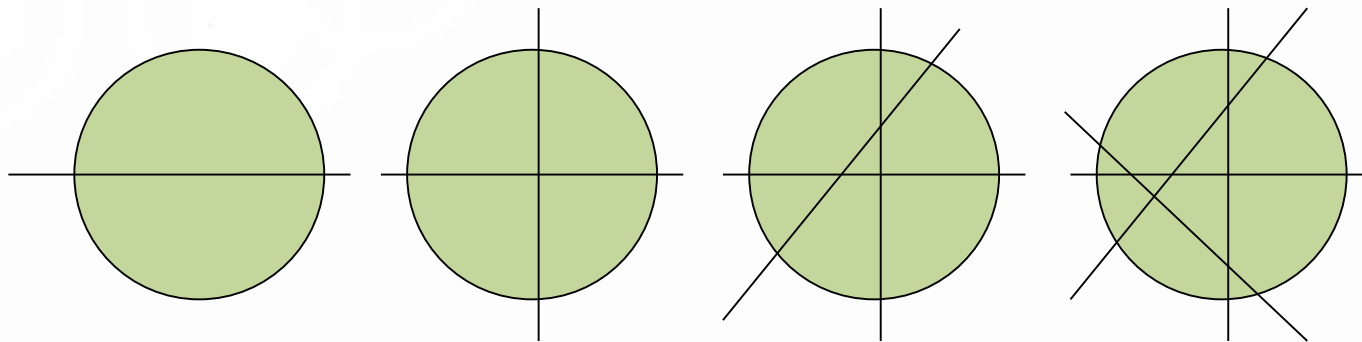




本节内容

- 什么是递归
- 深入理解递归的过程
- 递归的作用
 - ◆ 用递归来完成递推
 - ◆ 模拟连续发生的动作
 - ◆ 进行“自动的分析”

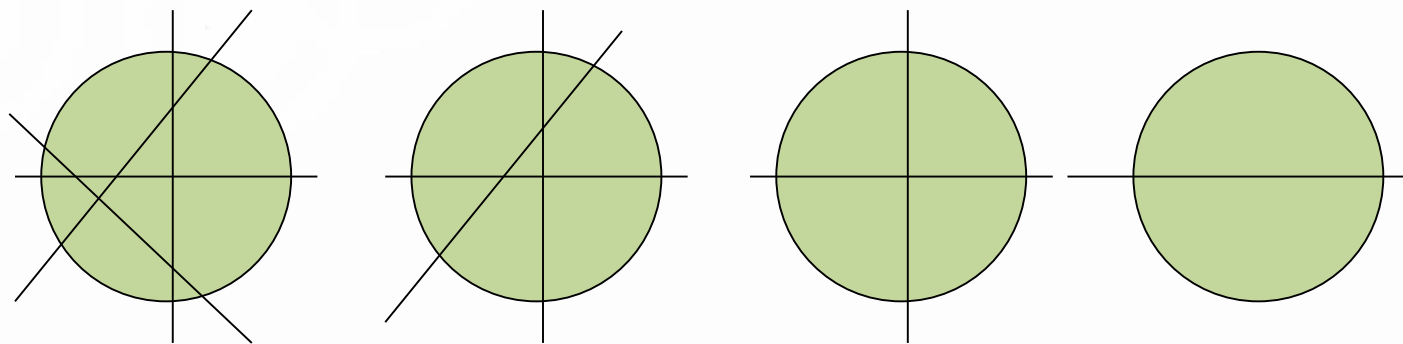
再从这个例子开始说起



- $q(0) = 1;$
- $q(1) = 1+1=2$
- $q(2) = 1+1+2=4;$
- $q(3) = 1+1+2+3=7;$
- $q(4) = 1+1+2+3+4=11;$
-
- $q(n) = q(n-1)+n;$

```
#include <iostream>
using namespace std;
int main()
{
    int blockCount = 0;
    int i = 0, N = 0;
    cin >> N;
    blockCount = 1;
    for (i = 1; i <= N; i++)
        blockCount = blockCount + i;
    cout << "最多可切" << blockCount << "块" << endl;
    return 0;
}
```

另一种解决方案



- $q(n) = q(n-1) + n;$
- $q(n-1) = q(n-2) + n-1$
- $q(n-2) = q(n-3) + n-2$
-
- $q(2) = q(1) + 2 = 4;$
- $q(1) = q(0) + 1 = 2$
- $q(0) = 1$

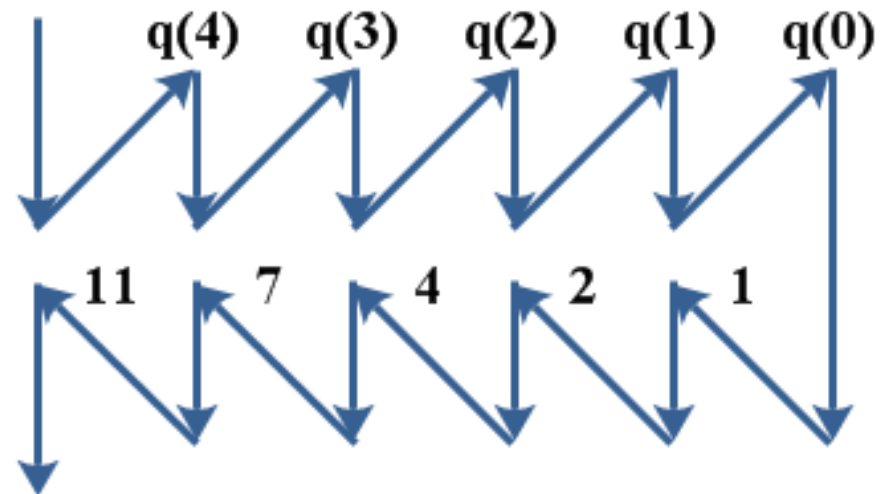
```
int q(int n)
{
    if (n == 0)
        return 1;
    else
        return(n + q(n - 1));
}
```

另一种解决方案

```
#include<iostream>
using namespace std;
int q(int n){
    if (n == 0)
        return 1;
    else
        return(n + q(n - 1));
}

int main(){
    cout << q(4) << endl;
    return 0;
}
```

Main()





递归与递推

■不同

- ◆ 递推的关注点放在 起始点 条件 而

- ◆ 递归的关注点放在 求解目标 上

■相同

- ◆ 重在表现 第 i 次 与 第 $i+1$ 次 的关系



用递归实现递推

■ 优点

- ◆ 让程序变得简明

■ 方法：

- ◆ 把关注点放在要求解的目标上

进而

- ◆ 找到第 n 次做与第 $n-1$ 次做之间的关系；
- ◆ 确定第 1 次的返回结果；

再看一例

■ 斐波那契数列

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

- $\text{fab}(n) = \text{fab}(n-1) + \text{fab}(n-2)$;
- $\text{fab}(1)=1, \text{fab}(2)=1$;

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



本节内容

- 什么是递归
- 深入理解递归的过程
- 递归的作用
 - ◆ 用递归来完成递推
 - ◆ 模拟连续发生的动作
 - ◆ 进行“自动的分析”

进制转换

- 将123转换成等值的二进制数：

除以2的商（取整） 余数

$$123/2 = 61 \quad 1$$

$$61/2 = 30 \quad 1$$

$$30/2 = 15 \quad 0$$

$$15/2 = 7 \quad 1$$

$$7/2 = 3 \quad 1$$

$$3/2 = 1 \quad 1$$

$$1/2 = 0 \quad 1$$

- 自下而上收集余数：1111011

```
void convert(int x)
{
    if ((x / 2) != 0)
    {
        convert(x / 2);
        cout << x % 2;
    }
    else
        cout << x;
}
```

进制转换

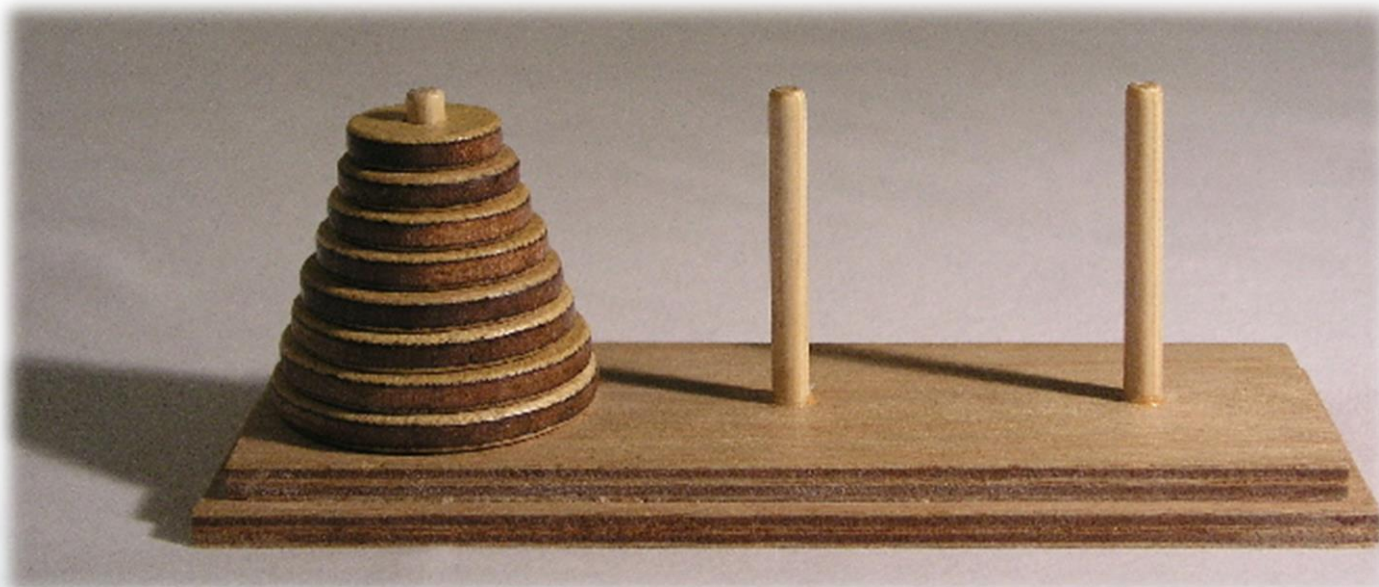
```
#include<iostream>
using namespace std;
void convert(int x)
{
    if ((x / 2) != 0)
    {
        convert(x / 2);
        cout << x % 2;
    }
    else
        cout << x;
}
int main()
{
    int x;
    cin >> x;
    convert(x);
    return 0;
}
```

123

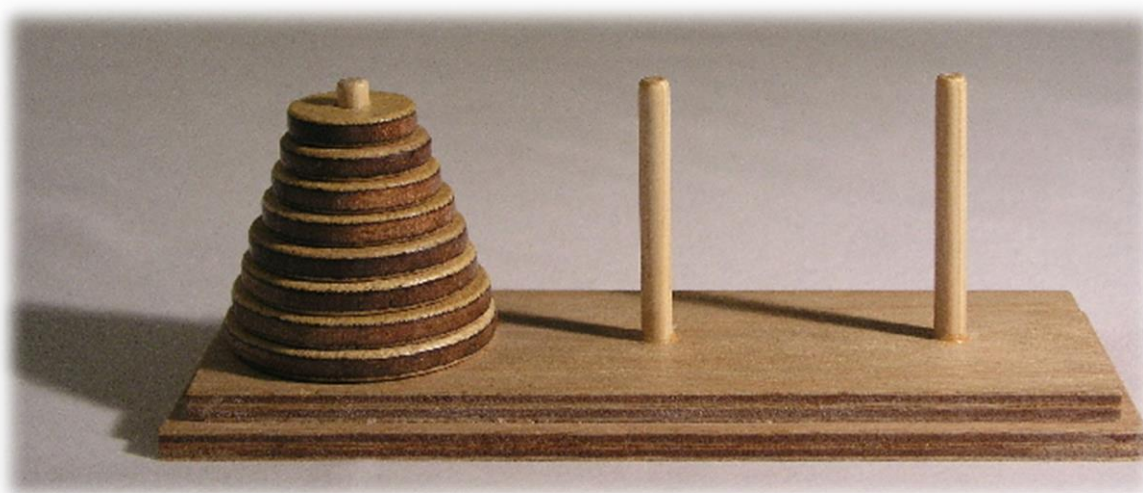
1111011

递归经典问题——汉诺塔问题

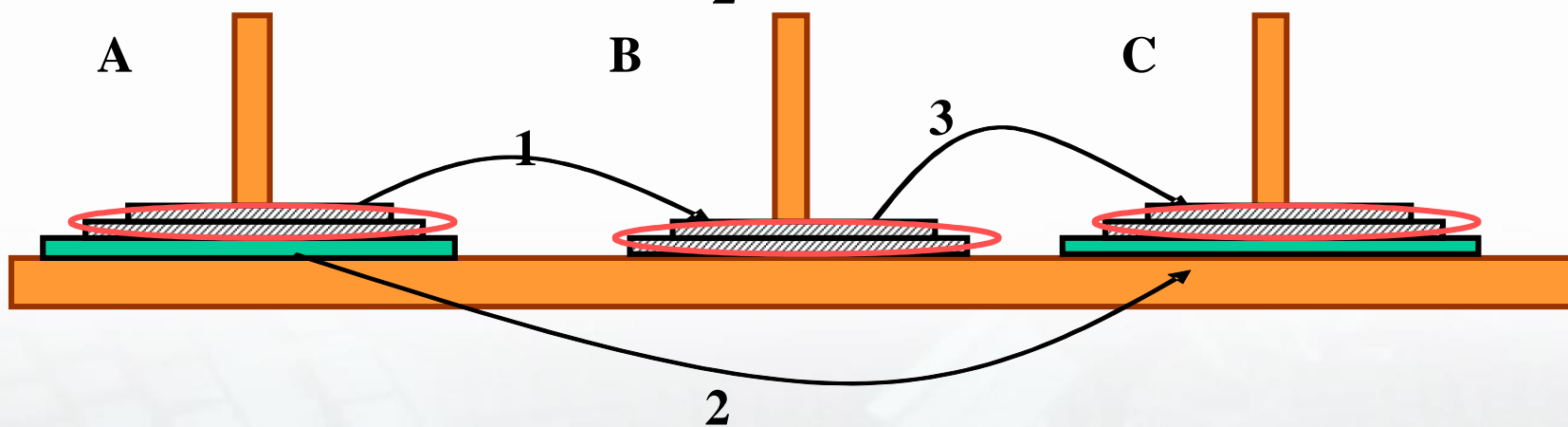
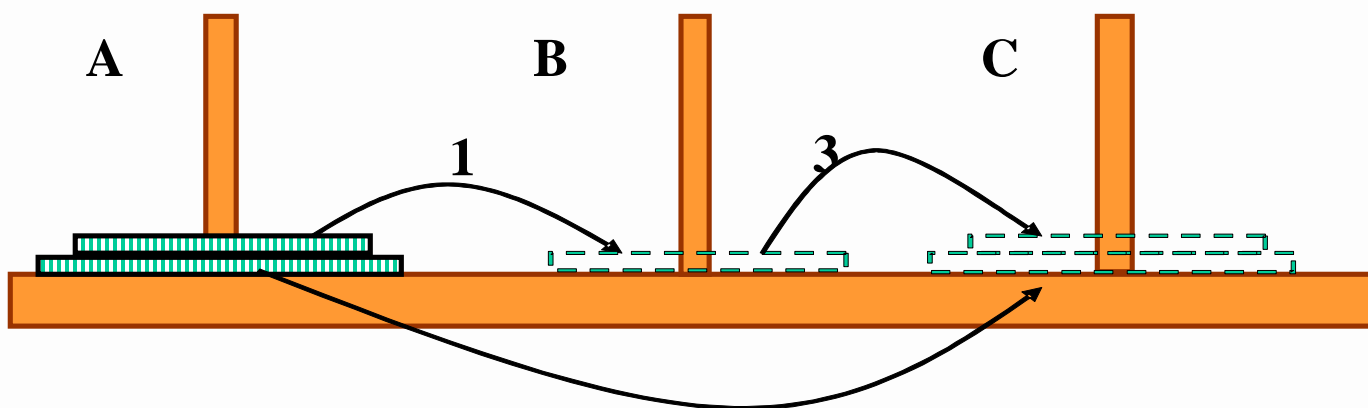
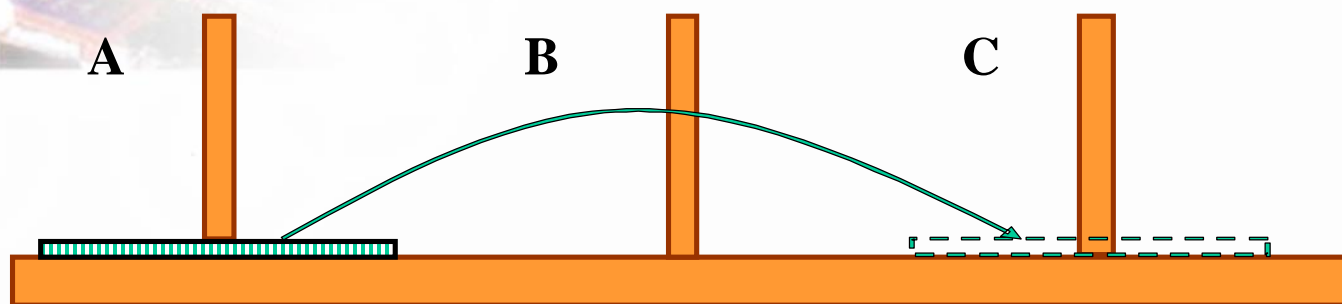
- 故事：相传在古代印度的Bramah庙中，有位僧人整天把三根柱子上的金盘倒来倒去，他想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中恪守下述规则：每次只允许移动一只盘，且大盘不得落在小盘上面。
- 有人会觉得这很简单，真的动手移盘就会发现，如以每秒移动一只盘子的话，按照上述规则将64只盘子从一个柱子移至另一个柱子上，所需时间约为5800亿年。



递归经典问题——汉诺塔问题



请输入盘数 $n = 4$
在3根柱子上移4只盘的步骤为：
把一个盘子从a移动到b
把一个盘子从a移动到c
把一个盘子从b移动到c
把一个盘子从a移动到b
把一个盘子从c移动到b
把一个盘子从c移动到a
把一个盘子从a移动到b
把一个盘子从b移动到c
把一个盘子从a移动到c
把一个盘子从b移动到a
把一个盘子从c移动到a
把一个盘子从b移动到c
把一个盘子从a移动到b
把一个盘子从c移动到b
把一个盘子从c移动到a



要实现 : $\text{move}(n, A, B, C)$

需进行 :

$\text{move}(n-1, A, C, B)$

move from A to C

$\text{move}(n-1, B, A, C)$

```
void move(int m, char x, char y, char z) //将m个盘子从A经过B移动到C
{
    if (m == 1)
    {
        cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
    }
    else
    {
        move(m - 1, x, z, y);
        cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
        move(m - 1, y, x, z);
    }
}
```

```

#include<iostream>
using namespace std;
void move(int m, char x, char y, char z)
{
    if (m == 1)
    {
        cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
    }
    else
    {
        move(m - 1, x, z, y);
        cout << "把一个盘子从 " << x << " 移动到 " << z << endl;
        move(m - 1, y, x, z);
    }
}

int main() {
    int n;
    cout << "请输入盘数n = ";
    cin >> n;
    cout << "在3根柱子上移" << n << "只盘的步骤为:" << endl;
    move(n, 'A', 'B', 'C');
    return 0;
}

```

```

请输入盘数n = 4
在3根柱子上移4只盘的步骤为:
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C
把一个盘子从 A 移动到 B
把一个盘子从 C 移动到 A
把一个盘子从 C 移动到 B
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C
把一个盘子从 B 移动到 A
把一个盘子从 C 移动到 A
把一个盘子从 B 移动到 C
把一个盘子从 A 移动到 B
把一个盘子从 A 移动到 C
把一个盘子从 B 移动到 C

```

模拟连续发生的动作

■ 方法

- ◆ 搞清楚 连续发生的动作是什么；

```
void move(int m, char x, char y, char z)
```

- ◆ 搞清楚 不同次动作之间的关系；

```
move(m - 1, x, z, y);  
cout << "把一个盘子从 " << x << " 移动到 " << z << endl;  
move(m - 1, y, x, z);
```

- ◆ 搞清楚 边界条件是什么；

```
if (m == 1)  
{  
    cout << "把一个盘子从 " << x << " 移动到 " << z << endl;  
}
```



本节内容

- 什么是递归
- 深入理解递归的过程
- 递归的作用
 - ◆ 用递归来完成递推
 - ◆ 模拟连续发生的动作
 - ◆ 进行“自动的分析”

逆波兰表达式

■ 题目描述

- ◆ 逆波兰表达式是一种把运算符前置的算术表达式:
 - 如 $2 + 3$ 的逆波兰表示法为 $+ 2 3$
 - 如 $(2 + 3) * 4$ 的逆波兰表示法为 $\times + 2 3 4$
- ◆ 编写程序求解任一仅包含 $+ - * /$ 四个运算符的逆波兰表达式的值。

■ 输入： $\times + 11.0 12.0 + 24.0 35.0$

■ 输出：1357.0

逆波兰表达式

$\times \div + 12 \ 36 + 1 \ 3 - 15 \ 8$

[*notation()*]

\times [*notation()*] [*notation()*]

\div [*notation()*] [*notation()*]

$+$ [*notation()*] [*notation()*]

$\times \rightarrow notation() \times notation()$

$\div \rightarrow notation() \div notation()$

$+$ $\rightarrow notation() + notation()$

$- \rightarrow notation() - notation()$

其他字符：将读到的字符串解释为数；

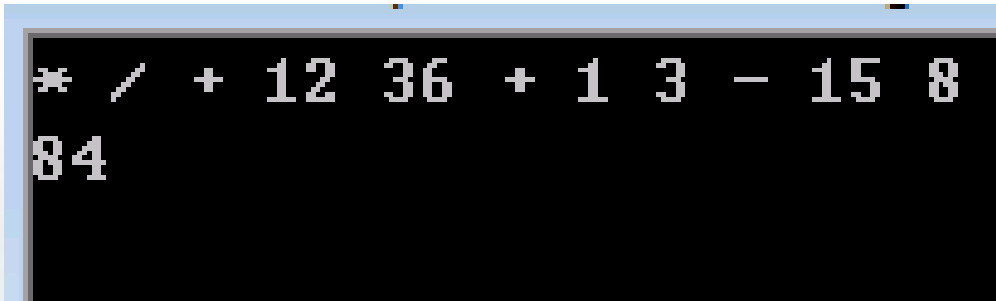
```
double notation()
{
    char str[10];
    cin >> str;
    switch (str[0])
    {
        case '+': return notation() + notation();
        case '-': return notation() - notation();
        case '*': return notation() * notation();
        case '/': return notation() / notation();
        default: return atof(str);
    }
}
```

逆波兰表达式

```
#include<iostream>
using namespace std;
double notation()
{
    char str[10];
    cin >> str;
    switch (str[0])
    {
        case '+': return notation() + notation();
        case '-': return notation() - notation();
        case '*': return notation() * notation();
        case '/': return notation() / notation();
        default: return atof(str);
    }
}
int main()
{
    cout << notation();
    return 0;
}
```

$\times \div + 12 \ 36 + 1 \ 3 - 15 \ 8$

$((12 + 36) \div (1 + 3)) \times (15 - 8)$



```
* / + 12 36 + 1 3 - 15 8
84
```



放苹果

■ 题目描述

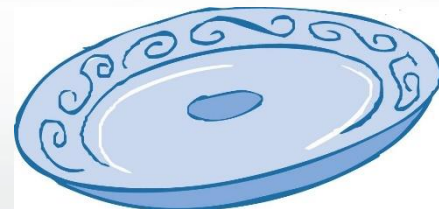
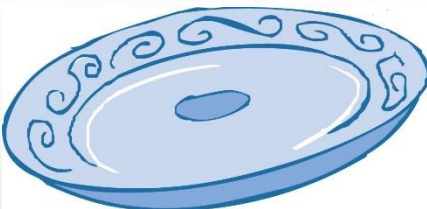
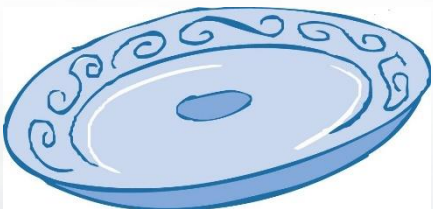
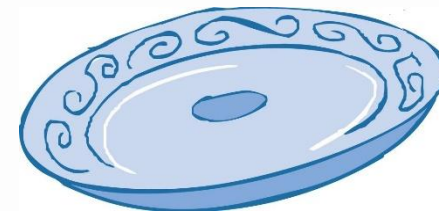
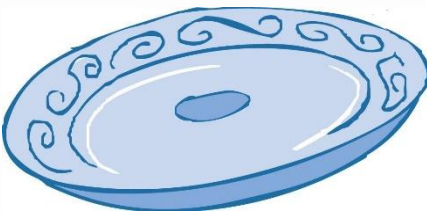
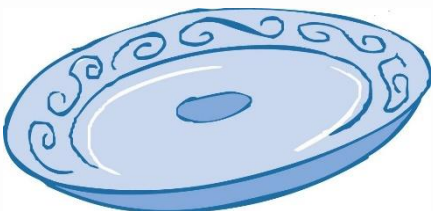
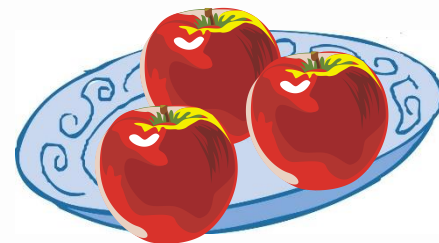
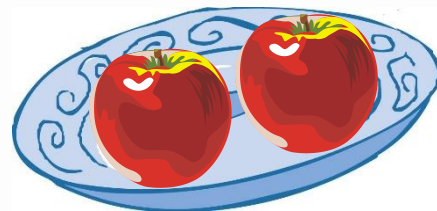
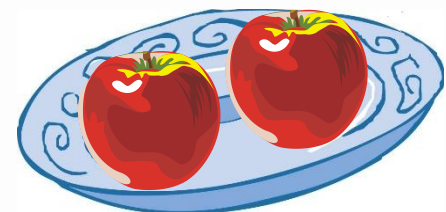
- ◆ 把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？
- ◆ 注意：5，1，1和1，5，1 是同一种分法
- ◆ 输入：7 3
- ◆ 输出：8

- 问题：M个苹果 放入 N个盘子，多少种放法
- 假设：有一个函数 $f(m, n)$ 能告诉我答案；

放苹果

如果： $n/\text{盘子数} > M/\text{苹果数}$

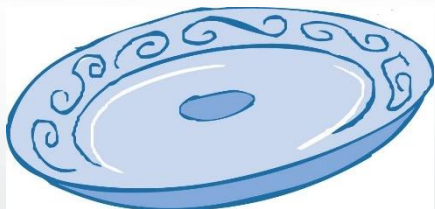
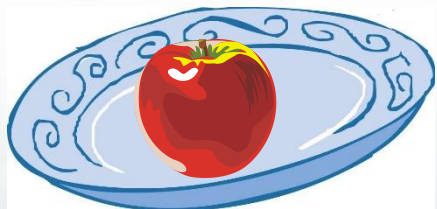
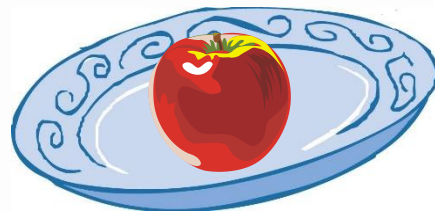
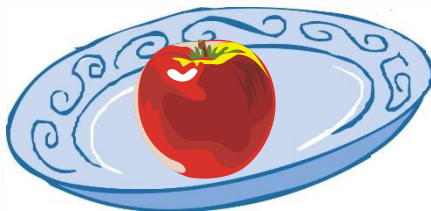
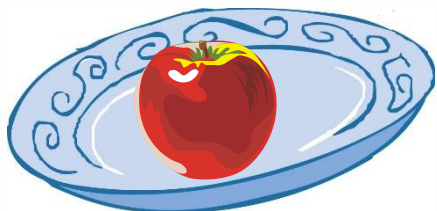
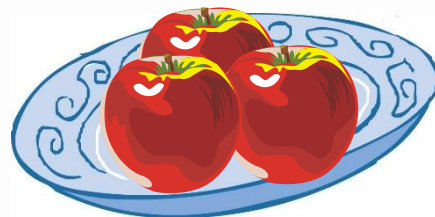
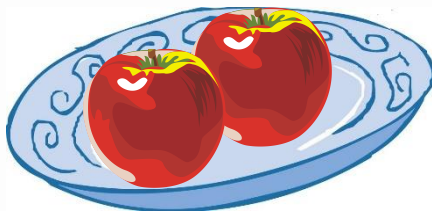
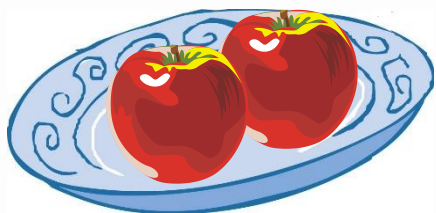
$$\text{if}(n > m) f(m, n) = f(m, m)$$



放苹果

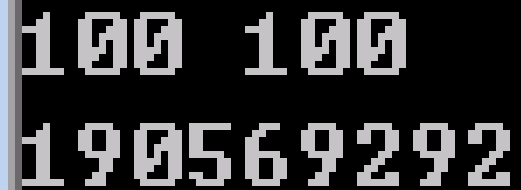
如果： $M/\text{苹果数} \geq n/\text{盘子数}$

$$\left\{ \begin{array}{l} \text{有盘子空着} : f(m, n) = f(m, n-1) \\ \text{没盘子空着} : f(m, n) = f(m-n, n) \end{array} \right.$$



放苹果

```
#include<iostream>
using namespace std;
int count(int m, int n)
{
    if (m <= 1 || n <= 0) return 1;
    if (m < n)
        return count(m, m);
    else
        return count(m, n - 1) + count(m - n, n);
}
void main()
{
    int apples, plates;
    cin >> apples >> plates;
    cout << count(apples, plates);
}
```



100 100
190569292

利用递归进行“自动分析”

■ 方法

- ◆ 先假设 有一个函数 能 给出答案

`notation()` `count(int m, int n)`

- ◆ 在利用 这个函数 的前提下，分析如何解决
问题；

```
'+' : return notation() + notation();  
 '-' : return notation() - notation();  
 '*' : return notation() * notation();  
 '/' : return notation() / notation();
```

```
if (m < n)  
    return count(m, m);  
else  
    return count(m, n - 1) + count(m - n, n);
```

- ◆ 搞清楚 最简单的情况下 答案 是什么.

```
default: return atof(str);
```

```
if (m <= 1 || n <= 1) return 1;
```



总结一下

■ 递归的作用

- ◆ 用递归来完成递推
- ◆ 模拟连续发生的动作
- ◆ 进行“自动的分析”



总结一下

■ 递归的作用

◆ 用递归来完成递推

● 方法：

- ◆ 把关注点放在要求解的目标上；
- ◆ 找到第 n 次与第 $n-1$ 次执行之间的关系；
- ◆ 确定第1次的返回结果；
- ◆ 模拟连续发生的动作
- ◆ 进行“自动的分析”



总结一下

■ 递归的作用

- ◆ 用递归来完成递推
- ◆ 模拟连续发生的动作

● 方法

- ◆ 搞清楚 连续发生的动作是什么；
- ◆ 搞清楚 不同次动作之间的关系；
- ◆ 搞清楚 边界条件是什么；
- ◆ 进行 “自动的分析”

总结一下

■ 递归的作用

- ◆ 用递归来完成递推
- ◆ 模拟连续发生的动作
- ◆ 进行“自动的分析”

● 方法

- ◆ 先假设 有一个函数 能 给出答案
- ◆ 在利用 这个函数 的前提下，分析如何解决问题；
- ◆ 搞清楚 最简单的情况下 答案 是什么.



好好想想,有没有问题？

谢 谢 ！