



中国科学技术大学

University of Science and Technology of China

# 程序设计 II

Programming Design II



递归



主讲：吴锋

# 目录

## CONTENTS

递归的定义

例题1：阶乘

例题2：汉诺塔

例题3：放苹果

例题4：神奇口袋

例题5：八皇后问题

例题6：黑瓷砖上行走

例题7：逆波兰表达式



# ◎ 递归的定义

- 一个过程或函数在其定义或说明中又直接（或间接）调用自身的一种方法。它通常把一个复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。
- 一般来说，递归需要有边界条件、递归策略和递归返回段。当边界条件不满足时，递归往边界条件前进；当边界条件满足时，递归返回。
- 递归求解的关键步骤：
  1. 找出递归的求解策略（递推公式）；
  2. 找到递归的终止条件（出口条件）；
  3. 先设计出递归函数的原型，然后在函数体中进行递归调用。



# ◎ 递归的应用

- 数据的定义是按递归定义的。
  - 例如：Fibonacci函数；阶乘函数等。
- 问题的解法按递归算法实现。
  - 这类问题虽则本身没有明显的递归结构，但用递归求解比迭代求解更简单，例如：汉诺塔（Hanoi）问题，八皇后问题。
- 数据的结构形式是按递归定义的。
  - 如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述。（《数据结构》、《算法基础》等课程涉及）



# ◎ 例题1：阶乘

- 问题描述

- 阶乘的递推定义：

$$0!=1, 1!=0!*1=1, 2!=1!*2=2$$

- 阶乘的递归定义：

$$n! = (n-1)! * n, \quad 0!=1.$$

- 如何用C语言函数求n的阶乘？



# ◎ 例题1：阶乘

- 求阶乘的递推程序

```
int fact (int n)
{
    if (n < 0)
        return -1;

    int fact = 1, i = 1;
    while (i <= n) {
        fact *= i;
        i++;
    }

    return fact;
}
```

## ◎ 例题1：阶乘

- 求阶乘的递归程序

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

→ 函数原型

} 出口条件

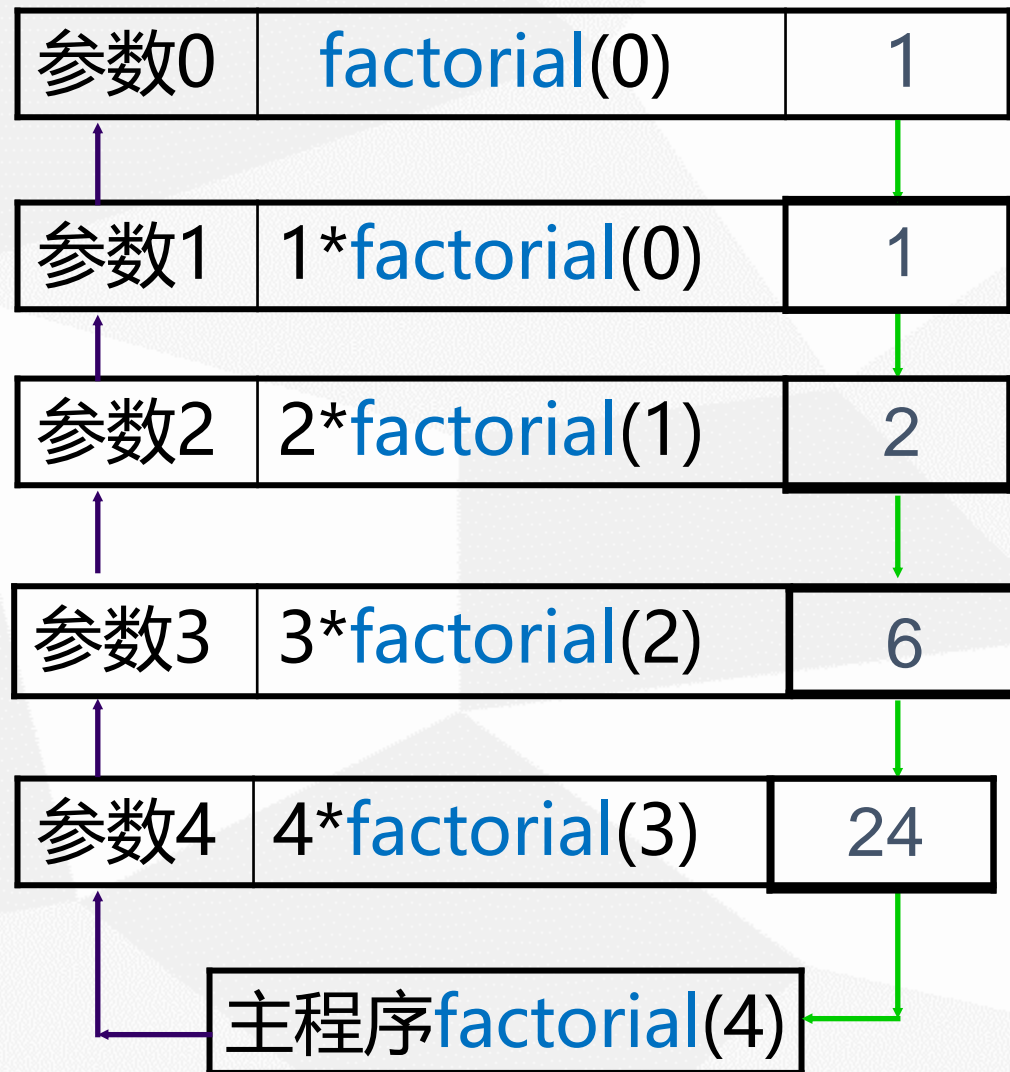
→ 递推公式



# 例题1：阶乘

```
int factorial (int n)
{
    // 递归返回
    if (n==0) return 1;

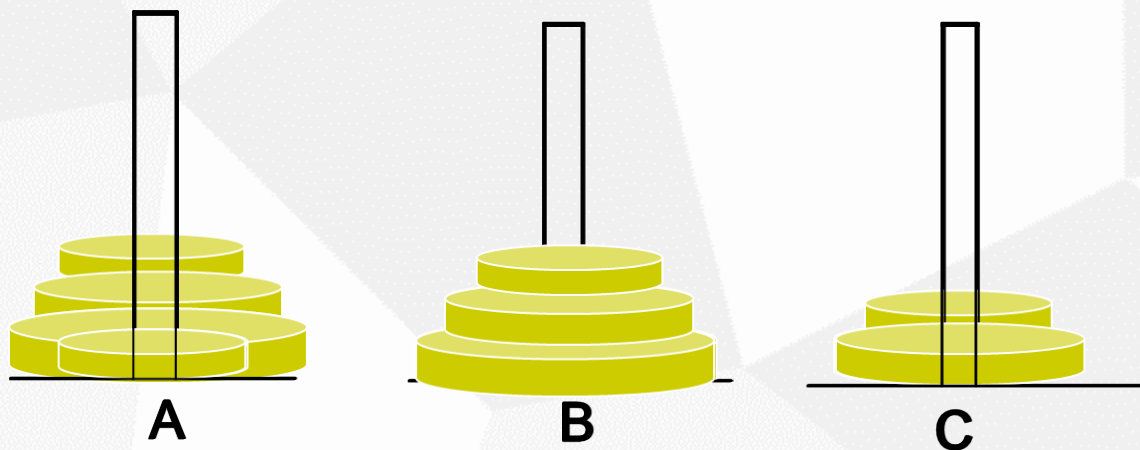
    // 递归策略
    return n*factorial(n-1);
}
```





## ◎ 例题2：汉诺塔

- **规则描述：**以C柱为中转站，将盘从A柱移动到B柱上，一次只能移动一个盘，而且大盘不能压在小盘上面。
- **求解任务：**写程序描述移动的过程，要求移动次数最少。



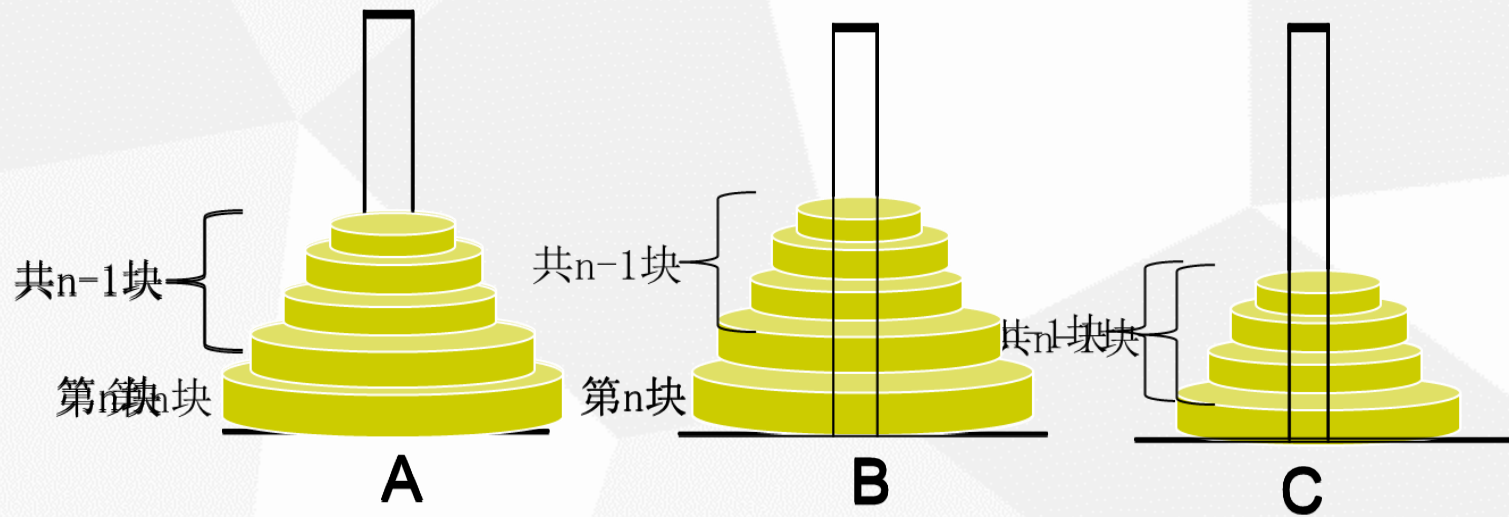
## ◎ 例题2：汉诺塔

- 问题1：如何描述移动的过程？
  - 对于这种典型的非数值计算的问题，可以用一个字符串来描述移动“指令”：
    - 例如：A->C
    - 表示把A柱最上面的一块盘移动到C柱最上面。
  - 3块盘的汉诺塔问题移动过程：
    - A->B, A->C, B->C, A->B, C->A, C->B, A->B



## ◎ 例题2：汉诺塔

- 问题2：如何应用递归求解？
  - 关键点一：递归的策略





## ◎ 例题2：汉诺塔

- 问题2：如何应用递归求解？

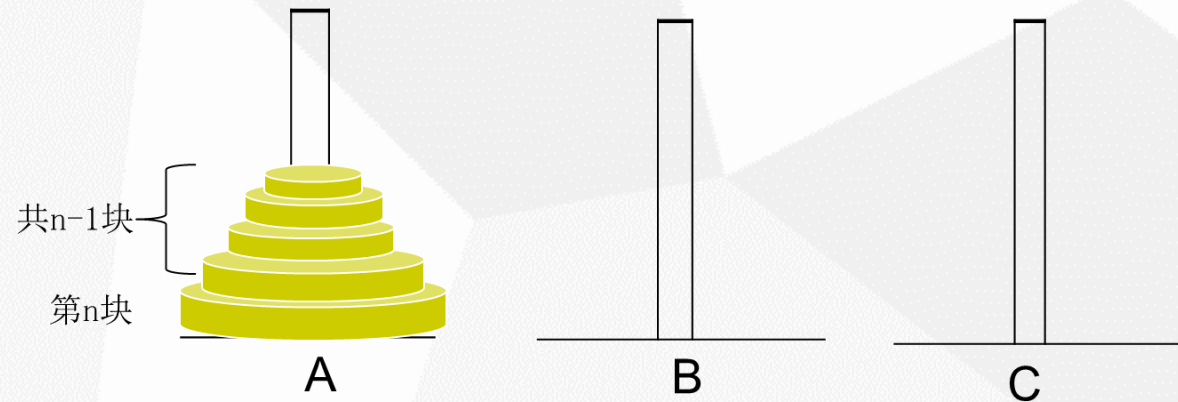
- 关键点一：递归的策略

- 将 $n$ 个盘从A柱移动到B柱，用C柱做中转，可分3步：

- ① 先将A上的 $n-1$ 个盘，以B为中转，从A柱移到C柱；

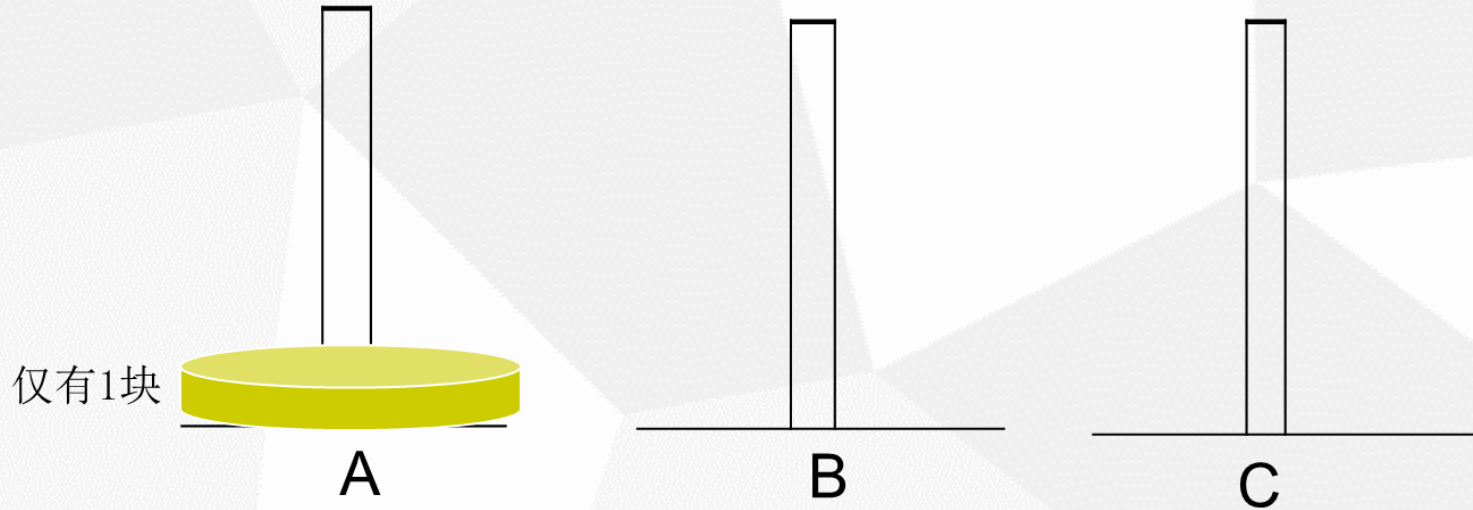
- ② 将A中最大的一个盘从A移动到B；

- ③ 将C柱上的 $n-1$ 个盘，以A为中转，移动到B柱。



## ◎ 例题2：汉诺塔

- 问题2：如何应用递归求解？
  - 关键点二：递归的结束条件
    - 当  $n==1$ ,  $A \rightarrow B$



## ◎ 例题2：汉诺塔

- 问题2：如何应用递归求解？
  - 关键点三：递归函数的原型

// 将 n 个盘子从 a 柱移动到b柱，用c柱做中转  
`void Hanoi(int n, char a, char b, char c)`

- 函数名：Hanoi;
- 函数参数：涉及到了int n, char a, b, c;
- 返回值：不需要，void





## ◎ 例题2：汉诺塔

```
#include <stdio.h>
```

```
//将 n 个盘子从 a 柱移动到b柱，用c柱做中转
```

```
void Hanoi(int n, char a, char b, char c);
```

```
int main() {
```

```
    int N;
```

```
    printf("Input disc number:\n");
```

```
    scanf("%d", &N);
```

```
    printf("The solution is:\n");
```

```
    Hanoi(N, 'A', 'B', 'C');
```

```
    return 0;
```

```
}
```

```
//将 n 个盘子从 a 柱移动到b柱，用c柱做中转
```

```
void Hanoi(int n, char a, char b, char c) {
```

```
    if (n == 1) {
```

```
        printf("%c->%c\n", a, b);
```

```
        return;
```

```
    }
```

```
//先将n-1个盘子，以b为中转，从a柱移动到c柱，
```

```
Hanoi(n-1, a, c, b);
```

```
//将一个盘子从a移动到b
```

```
printf("%c->%c\n", a, b);
```

```
//将c柱上的n-1个盘子，以a为中转，移动到b柱
```

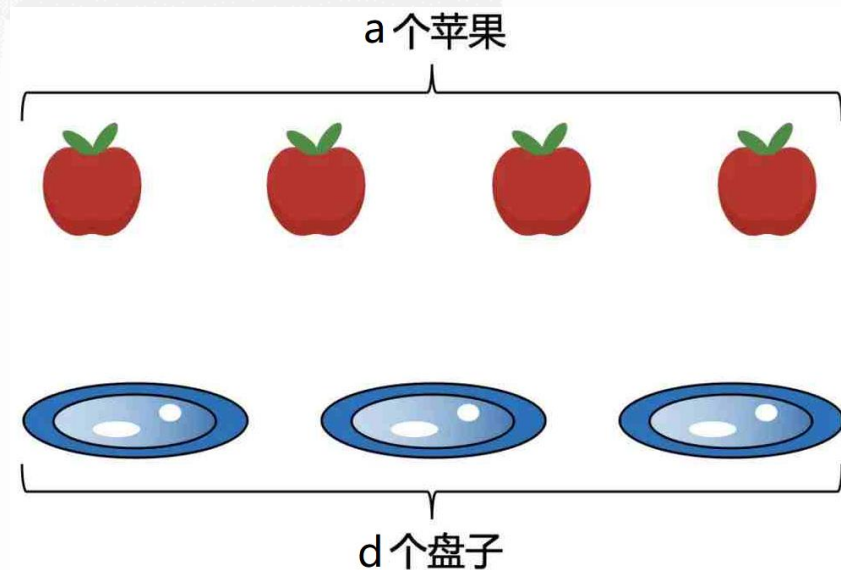
```
Hanoi(n-1, c, b, a);
```

```
}
```

## ◎ 例题3：放苹果

### • 问题描述 (P178)

- a个苹果，d个盘子，问多少种不同放法。
- 盘子和苹果都是没有编号的。
  - 即 1,2,3 和 3,2,1算是同一种放法。
- 设  $f(a,d)$  为a个苹果，d个盘子的放法数目。



## ◎ 例题3：放苹果

### • 解题思想：先对 $a$ 作讨论

◦ 如果  $d > a$ ，必定至少有  $d - a$  个盘子是空的。

• 去掉这些空盘子对摆放苹果方法数目不产生影响，即：

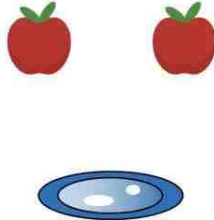
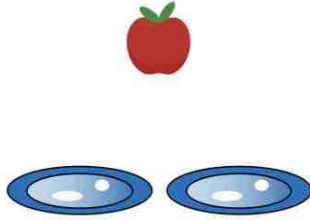
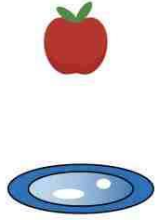
$$\text{if } (d > a) \ f(a, d) = f(a, a)$$

◦ 当  $d \leq a$  时，不同的放法可以分成两类：

1. 有盘子空着，假定该情况下的放法数目为  $g(a, d)$

2. 所有盘子都有苹果，即没有盘子空着，假定该情况下的放法数目为  $k(a, d)$

• 则显然： $f(a, d) = g(a, d) + k(a, d)$

		
苹果比盘子多， $a > d$	苹果比盘子少， $a < d$	苹果和盘子相同， $a = d$



## ◎ 例题3：放苹果

- 解题思想：先对a作讨论

- 当  $d \leq a$  时，不同的放法可以分成两类：

1. 有盘子空着，必定至少有1个盘子没有苹果，相当于在把a个苹果放在剩下的d-1个盘子里，即：  $g(a, d) = f(a, d-1)$
2. 所有盘子都有苹果，相当于先在所有的d的盘子都摆上1个苹果，然后再将剩下的a-d个苹果摆到d个盘子上，即：  $k(a, d) = f(a-d, d)$

- 因此：  $f(a, d) = f(a, d-1) + f(a-d, d)$

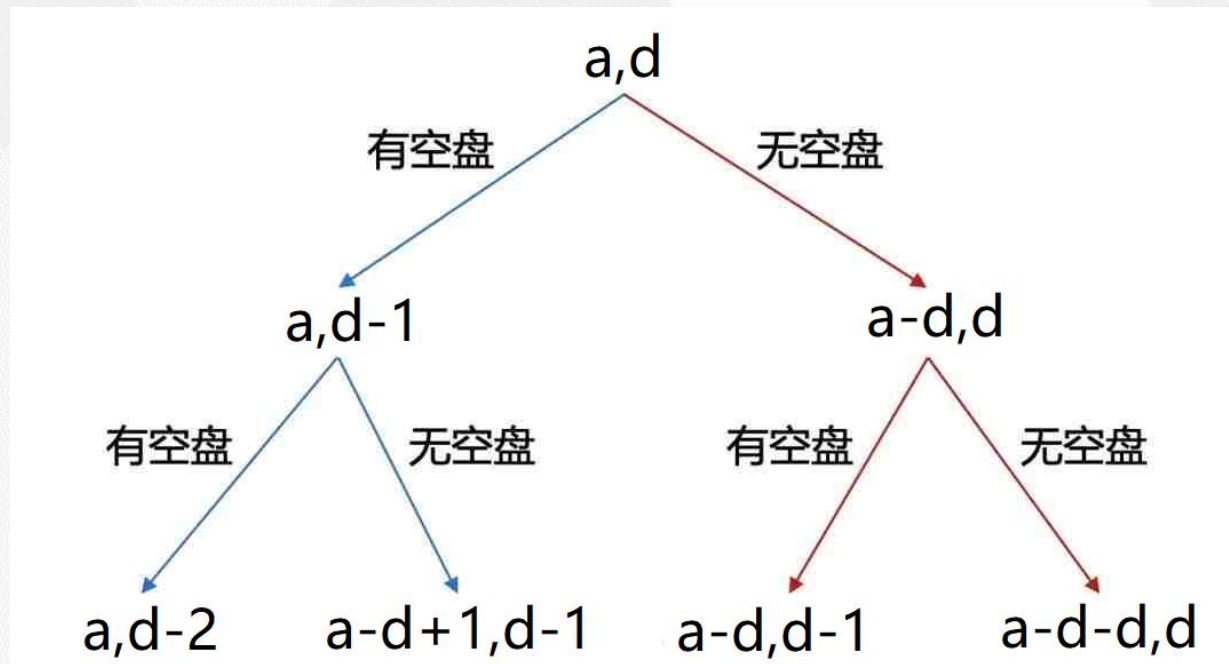
## ◎ 例题3：放苹果

### • 解题思路

◦ 综述，得到递推公式：

if  $(d > a)$   $f(a, d) = f(a, a)$ ;

else  $f(a, d) = f(a, d-1) + f(a-d, d)$ ;



## ◎ 例题3：放苹果

### • 解题思路

#### ◦ 递归的出口条件

- 当 $d=1$ 时，所有苹果都必须放在一个盘子里，所以返回1；
- 当没有苹果可放时，所有盘子都是空的，这当然是一种放法，所以也返回1；

#### ◦ 递归的两条路线：

- 第一条会逐渐减少，终会到达出口 $d==1$ ；
- 第二条 $a$ 会逐渐减少，因为 $d>a$ 时，我们会`return f(a,a)` 所以终会到达出口 $a==0$ 。





## ◎ 例题3：放苹果

```
int f(int a, int d)
{
    if (d == 1 || a == 0)
        return 1;

    if (d > a )
        return f(a, a);
    else
        return f(a, d-1) + f(a-d, d);
}
```

## ◎ 例题4：神奇口袋

- 问题描述

- 给出小于40的正整数 $a_1, a_2, \dots, a_n$ ，问凑出和为40有多少种方法。

- 解题思路

- 设  $f(a_1, a_2, \dots, a_n, 40)$  为用 $a_1, a_2, \dots, a_n$ 凑出40的方法数，考虑凑数的方法分为两类，用到 $a_1$ 和用不到 $a_1$ ，则有：

$$f(a_1, a_2, \dots, a_n, 40) = f(a_1, a_2, \dots, a_n, 40 - a_1) + f(a_2, \dots, a_n, 40)$$

## ◎ 例题4：神奇口袋

### • 解题思路

- 考虑到在实现上函数的参数个数要统一，所以把 $a_1, a_2, \dots, a_n$ 放到一个数组中，用`numbers[]` 存放所有的整数，`n`表示数组中整数的个数，`ith`表示从数组中第`ith`个数开始凑数(`ith`前面的不用)，`sum`表示要凑出的总数，则：

`f(numbers, n, ith, sum) =`

`f(numbers, n, ith+1, sum-numbers[ith])`    // 用到`ith`  
`+ f(numbers, n, ith+1, sum)`                      // 不用`ith`

## ◎ 例题4：神奇口袋

- 解题思路：出口条件

1. 如果  $\text{sum} == 0$ , 则应该返回 1

- 一个数都不取，也是一种取法

2. 如果  $\text{sum} < 0$  则应该返回 0

- 没法凑出负数

3. 如果  $\text{ith} == n$  则应该返回 0

- 所有的数都用完了，和还没有凑够

- 思考：1,3的判断先后次序是不是无所谓的？





## ◎ 例题4：神奇口袋

- 具体实现

- 在实现上有两种考虑：

- 因为numbers, n与递归没有太大关系，可以考虑使用全局量；
    - 如果不使用全局量，而通过参数传递numbers, n，可以增强递归函数的独立性。

```
int numbers[50], n; // 将numbers, n设为全局量
// 将 n 个数放入数组 numbers后, count(0, 40); 就能求出结果
int count(int ith, int sum)
{
    if (sum == 0) return 1;
    if (ith==n || sum <0) return 0;

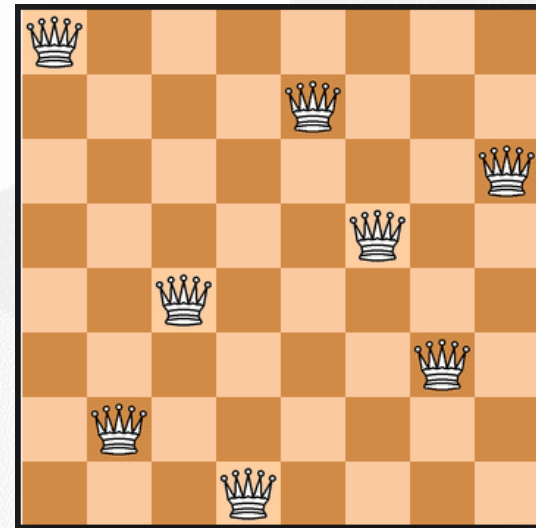
    return count(ith+1, sum - numbers[ith]) + count(ith+1, sum);
}
```



## ◎ 例题5：八皇后问题

### • 问题描述（P181）

- 根据国际象棋的规则，皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有 $8 * 8$ 个方格），使它们谁也不能被吃掉！这就是著名的8皇后问题。已经知道8皇后问题一共有92组解。
- 对于某个满足要求的8皇后的摆放方法，定义一个皇后串 $a$ 与之对应，即 $a=b_1b_2...b_8$ ，其中 $b_i$ 为相应摆法中第 $i$ 行皇后所处的列数。串的比较规则：皇后串 $x$ 置于皇后串 $y$ 之前，当且仅当将 $x$ 视为整数时比 $y$ 小。
- 给出一个数 $b$ ，要求输出第 $b$ 个串。



## ◎ 例题5：八皇后问题

- 输入：第1行是测试数据的组数 $n$ ，后面跟着 $n$ 行输入。每组测试数据占1行，包括一个正整数 $b$  ( $1 \leq b \leq 92$ )。
- 输出：输出有 $n$ 行，每行输出对应一个输入。输出应是一个正整数，是对应于 $b$ 的皇后串。

- 样例输入

2

1

92

- 样例输出

15863724

84136275



## ◎ 例题5：八皇后问题

### • 解题思路

- 8皇后问题可用8重循环解决，但是 $n$ 皇后问题呢？
- 递归可以用来实现任意多重循环。
- 如果有多个变量，每个变量有各自的取值范围，要覆盖这些变量值的全部组合，可以用递归实现。
- 此题没有什么直接的递推关系，写递归就是为了起到多重循环的作用。





## ◎ 例题5：八皇后问题

```
#include <stdio.h>
#include <math.h>
#include <string.h>
const int QueenNum = 8;
int anResult[92][QueenNum]; //存放找到的摆法
int anQueen[QueenNum]; //纪录当前正在尝试的摆法
int nFoundNum = 0; //当前已经找到多少种摆法
void Queen(int n); //摆放第n行以及以后的皇后(行号从0开始算)
int main() {
    Queen(0);

    int n, b;
    scanf("%d", &n);
    while (n--) {
        scanf("%d", &b);
        for (int i = 0; i < QueenNum; i++)
            printf("%d", anResult[b-1][i]+1);
        printf("\n");
    }
    return 0;
}
```



## 例题5：八皇后问题

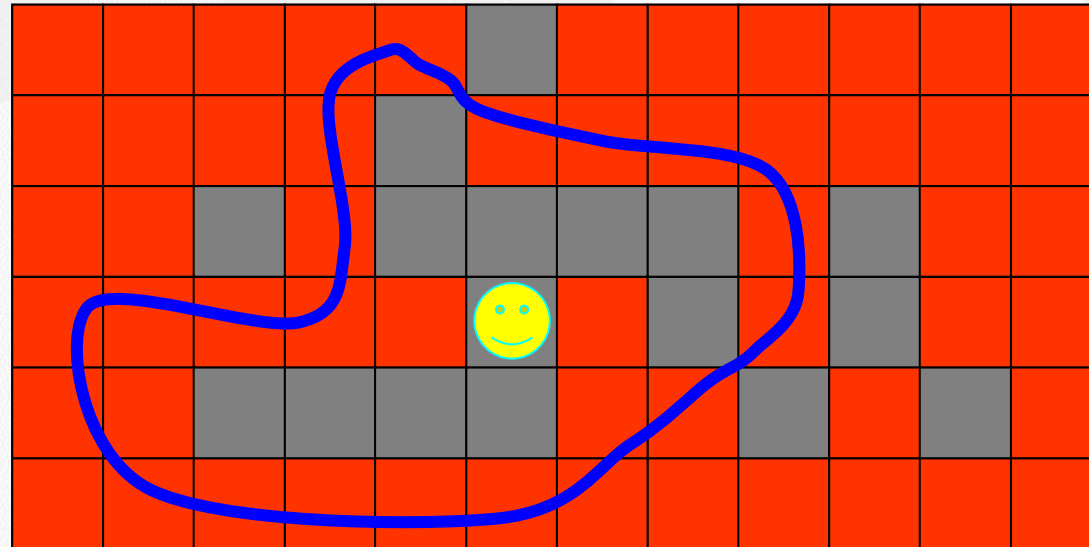
//摆放第n行以及以后的皇后(行号从0开始算)

```
void Queen(int n) {  
    if (n == QueenNum) { //前QueenNum行都成功摆好了，记下摆法  
        memcpy(anResult[nFoundNum++], anQueen, sizeof(anQueen));  
        return;  
    }  
    for (int i = 0; i < QueenNum; i++) { //尝试第n行所有位置  
        int j;  
        for (j = 0; j < n; j++) {  
            //对每个位置，判断是否和已经摆好的皇后冲突  
            if (i == anQueen[j] || abs(i - anQueen[j]) == abs(n - j))  
                break;  
        }  
        if (j == n) {  
            //如果没有冲突，则第n行摆好了，记下来，再摆第n+1行 anQueen[n] = i;  
            Queen(n+1);  
        }  
    }  
}
```

## ◎ 例题6：黑瓷砖上行走

### • 问题描述（P179）

- 有一间长方形的房子，地上铺了红色、黑色两种颜色的正方形瓷砖。你站在其中一块黑色的瓷砖上，只能向相邻的黑色瓷砖移动。请写一个程序，计算你总共能够到达多少块黑色的瓷砖。



## ◎ 例题6：黑瓷砖上行走

- 输入：

- 包括多个数据集合。每个数据集合的第一行是两个整数W和H，分别表示x方向和y方向瓷砖的数量。W和H都不超过20。在接下来的H行中，每行包括W个字符。每个字符表示一块瓷砖的颜色，规则如下
  - ‘.’：黑色的瓷砖； ‘#’：红色的瓷砖；
  - ‘@’：黑色的瓷砖，并且你站在这块瓷砖上。该字符在每个数据集合中唯一出现一次。
  - 当在一行中读入的是两个零时，表示输入结束。

- 输出

- 对每个数据集合，分别输出一行，显示你从初始位置出发能到达的瓷砖数(记数时包括初始位置的瓷砖)。





## ◎ 例题6：黑瓷砖上行走

• 样例输入：

6 9

....#.

.....#

.....

.....

.....

.....

.....

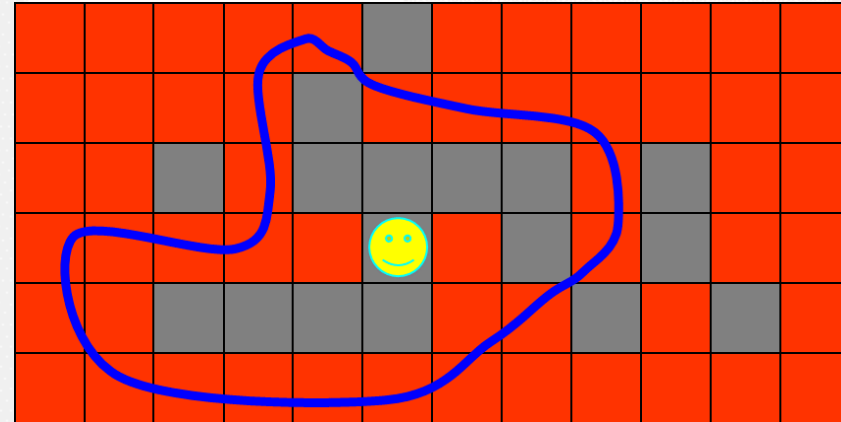
#@...#

.#..#.

0 0

• 样例输出：

45



## ◎ 例题6：黑瓷砖上行走

### • 解题思路

- 设  $f(x, y)$  为从点  $(x, y)$  出发能够走过的黑瓷砖总数，而且  $(x, y)$  是黑瓷砖，则有：

$$f(x, y) = 1 + f(x-1, y) + f(x+1, y) \\ + f(x, y-1) + f(x, y+1)$$

- 基本思路就是从当前位置，不断的向左、右、上、下四个相邻的瓷砖进行试探。
- 这里需要注意，凡是走过的瓷砖不能够被重复走过。
- 递归终止条件：
  - 如果  $(x, y)$  是红瓷砖，则返回 0。



## 例题6：黑瓷砖上行走

```
#include <stdio.h>
#define MAX 22
char rect[MAX][MAX]; //方块四周加红色块，去掉边界判断，使得递归统一终止于红色块
int walkFrom(int currentRow, int currentCol); //返回从某点出发能走到的格子数
int main() {
    int col, row;
    while (scanf("%d %d", &col, &row) && col != 0 && row != 0){
        int i, j, startRow, startCol;
        for (i = 0; i < MAX; i++)
            for (j = 0; j < MAX; j++) rect[i][j] = '#'; //初始化为红砖
        for (i = 1; i <= row; i++)
            for (j = 1; j <= col; j++) {
                scanf("%c", &rect[i][j]);
                if (rect[i][j] == '@') {
                    startRow = i; startCol = j; rect[i][j]='.'; //人站立处是黑砖
                }
            }
        printf("%d\n", walkFrom(startRow, startCol));
    }
    return 0;
}
```

## 例题6：黑瓷砖上行走

```
int walkFrom(int currentRow, int currentCol)
{
    if(rect[currentRow][currentCol] == '#')
        return 0;
    else //本瓷砖算走过，以后不能再走了，设为红砖
        rect[currentRow][currentCol] = '#';

    return 1 + walkFrom(currentRow+1, currentCol)
        + walkFrom(currentRow-1, currentCol)
        + walkFrom(currentRow, currentCol+1)
        + walkFrom(currentRow, currentCol-1);
}
```





## ◎ 例题7：逆波兰表达式

### • 问题描述 (P176)

- 逆波兰表达式是一种把运算符前置的算术表达式
  - 例如普通的表达式  $2 + 3$  的逆波兰表示法为  $+ 2 3$ 。
- 逆波兰表达式的优点是运算符之间不必有优先级关系，也不必用括号改变运算次序
  - 例如：  $(2 + 3) * 4$  的逆波兰表示法为  $* + 2 3 4$ 。
- 本题求解逆波兰表达式的值，其中运算符包括  $+ - * /$  四个。



## ◎ 例题7：逆波兰表达式

- 输入数据

- 输入为一行，其中运算符和运算数之间都用空格分隔，运算数是浮点数。

- 输出要求

- 输出为一行，表达式的值。

- 输入样例

\* + 11.0 12.0 + 24.0 35.0

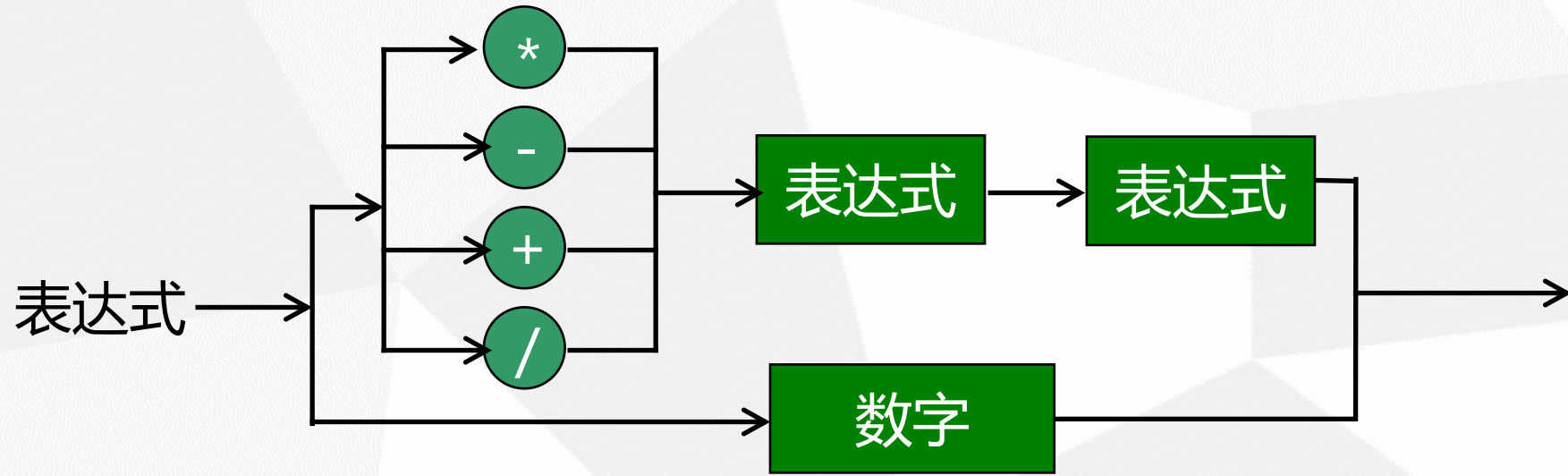
- 输出样例

1357.000000



## ◎ 例题7：逆波兰表达式

- 逆波兰表达式的递归定义



## 例题7：逆波兰表达式

```
#include <stdio.h>
#include<math.h>
double exp() {
    char a[10];
    scanf("%s", a);

    switch (a[0]) {
        case '+': return exp( ) + exp( );
        case '-': return exp( ) - exp( );
        case '*': return exp( ) * exp( );
        case '/': return exp( ) / exp( );
        default: return atof(a); //字符串转浮点数
    }
}

int main() {
    double ans = exp();
    printf("%f", ans);
    return 0;
}
```





## 例题7：逆波兰表达式

- 问题拓展一：改写此程序，要求将逆波兰表达式转换成常规表达式输出。可以包含多余的括号。

```
void exp() {  
    char a[10];  
    scanf("%s", a);  
  
    switch (a[0]) {  
        case '+': printf("("); exp(); printf(")"); printf("+");  
                  printf("("); exp(); printf(")"); break;  
        case '-': printf("("); exp(); printf(")"); printf("-");  
                  printf("("); exp(); printf(")"); break;  
        case '*': printf("("); exp(); printf(")"); printf("*");  
                  printf("("); exp(); printf(")"); break;  
        case '/': printf("("); exp(); printf(")"); printf("/");  
                  printf("("); exp(); printf(")"); break;  
        default: printf("%s", a); return;  
    }  
}  
  
int main() { exp(); return 0; }
```



## ◎ 例题7：逆波兰表达式

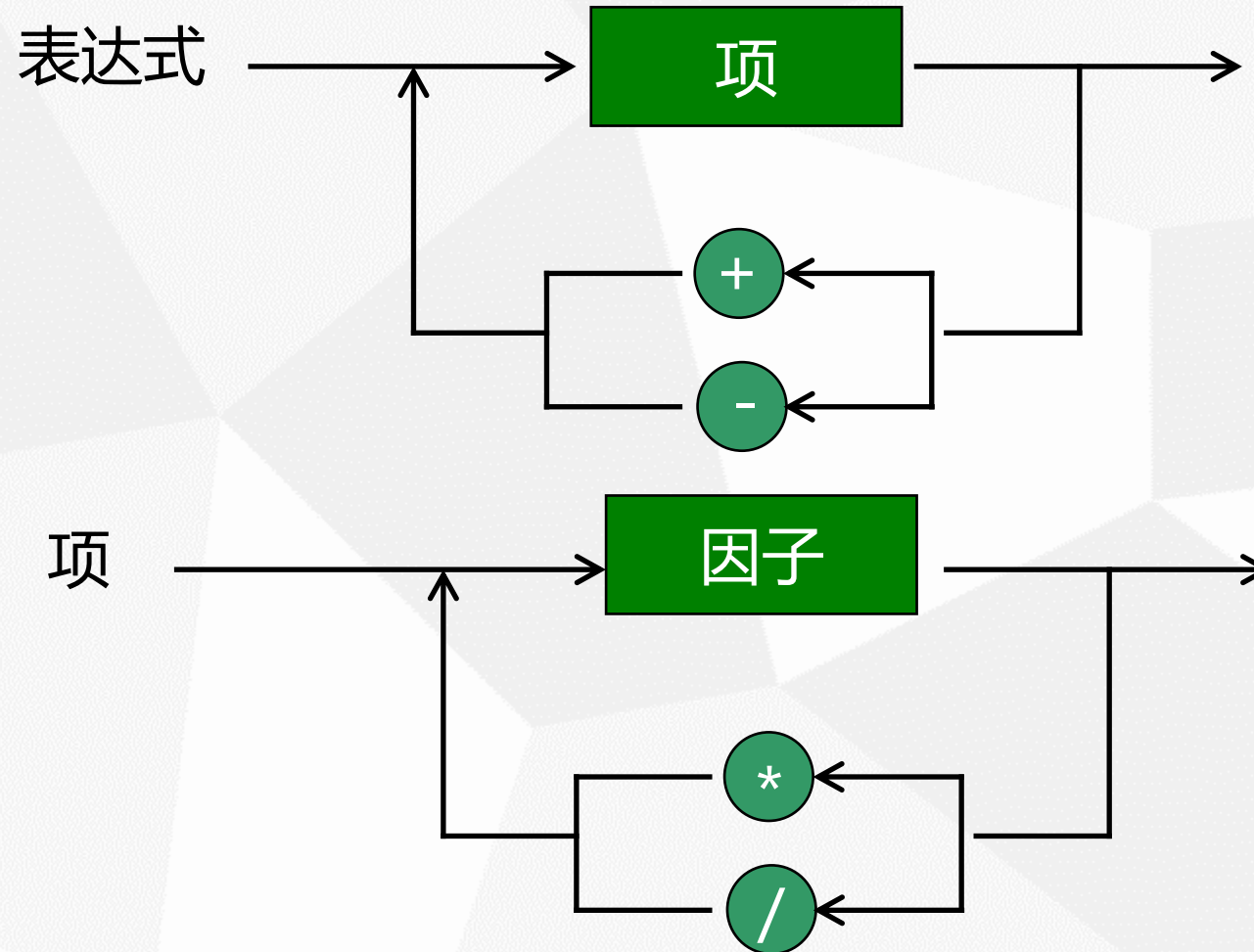
- 问题拓展二：常规表达式计算

- 输入为四则运算表达式，仅由数字、+、-、\*、/、(、)组成，没有空格，要求求其值。
- 解题思路：常规表达式也有一个递归的定义，因此对于表达式可以进行递归分析处理。



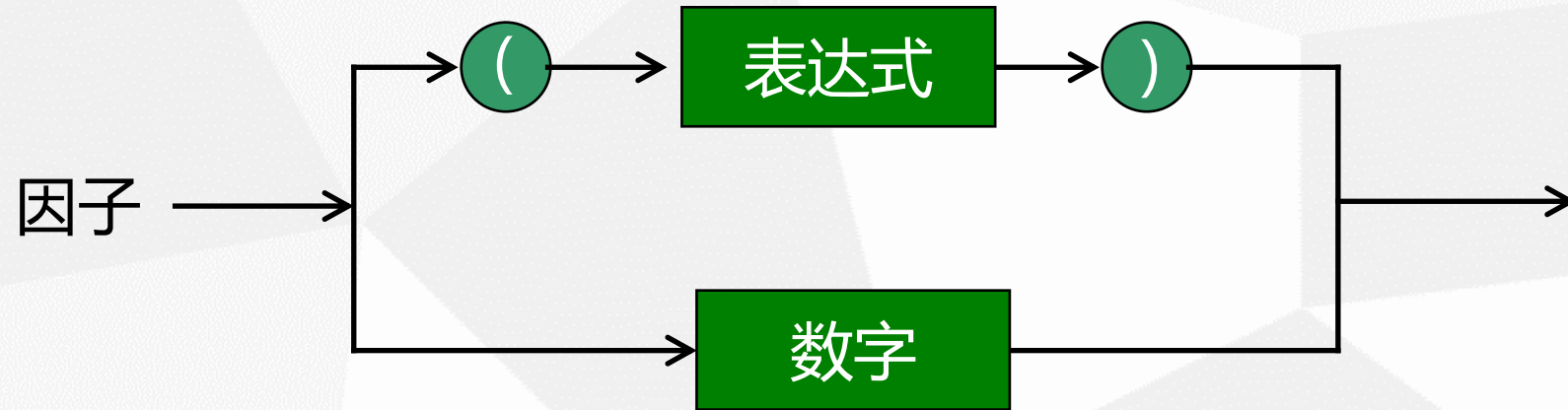
## ◎ 例题7：逆波兰表达式

- 常规表达式的递归定义



## ◎ 例题7：逆波兰表达式

- 常规表达式的递归定义





## ◎ 例题7：逆波兰表达式

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
char curr_char;
double factor_value();
double term_value();
double expression_value();

int main() {
    printf("Enter an expression: ");
    curr_char = getchar();
    double result = expression_value();
    printf("The result is %f\n", result);
    return 0;
}
```



## ◎ 例题7：逆波兰表达式

//求一个表达式的值

```
double expression_value() {  
    double result = term_value(); //求第一项的值  
    bool more = true;  
    while (more) {  
        char op = curr_char;  
        if (op == '+' || op == '-') {  
            curr_char = getchar(); //取下一个字符  
            int value = term_value();  
            if (op == '+') result += value;  
            else result -= value;  
        } else more = false;  
    }  
    return result;  
}
```



## 例题7：逆波兰表达式

//求一个项的值

```
double term_value() {  
    double result = factor_value(); //求第一个因子的值  
    bool more = true;  
    while (more) {  
        char op = curr_char;  
        if (op == '*' || op == '/') {  
            curr_char = getchar();  
            int value = factor_value();  
            if (op == '*') result *= value;  
            else result /= value;  
        } else more = false;  
    }  
    return result;  
}
```



## ◎ 例题7：逆波兰表达式

//求一个因子的值

```
double factor_value() {  
    double result = 0;  
    char c = curr_char;  
    if (c == '(') {  
        curr_char = getchar();  
        result = expression_value();  
        curr_char = getchar();  
    } else  
        while(isdigit(c)) {  
            result = 10 * result + c - '0';  
            curr_char = getchar(); c = curr_char;  
        }  
    return result;  
}
```





## ◎ 小结：递归的优缺点

- 运用递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。
- 递归方法解题运行效率较低（例如：用递推求  $n!$ ）。因此，应该尽量避免使用递归，除非没有更好的算法。
- 在递归调用的过程当中系统为每一层调用的返回点、局部变量等开辟了栈来存储（又称活动记录栈）。递归次数过多容易造成栈溢出等。
  - 注意：由于函数的局部变量是存在栈上的，如果有体积大的局部变量，比如数组，而递归层次又可能很深的情况下，也许会导致栈溢出，因此可以考虑使用全局数组或动态分配数组。



# ◎ 作业

## • 1. 城堡 (P189)

- 给定一个城堡的地形图。请你编写一个程序，计算城堡一共有多少房间，最大的房间有多大。城堡被分割成  $m * n$  ( $m \leq 50$ ,  $n \leq 50$ ) 个方块，每个方块可以有 0~4 面墙。

```

      1   2   3   4   5   6   7
#####
1 #   |   #   |   #   |   |   #
#####---#####---#---#####---#
2 #   #   |   #   #   #   #   #
#---#####---#####---#####---#
3 #   |   |   #   #   #   #   #
#---#####---#####---#---#
4 #   #   |   |   |   |   #   #
#####

```

```

#   = Wall
|   = No wall
-   = No wall

```

## • 2. 分解因数 (P189)

- 给出一个正整数 $a$ ，要求分解成若干个正整数的乘积，即 $a = a_1 * a_2 * a_3 * \dots * a_n$ ，并且 $1 < a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$ ，问这样的分解的种数有多少。注意到 $a = a$ 也是一种分解。

## • 3. 迷宫 (P189)

- 一天Extense 在森林里探险的时候不小心走入了一个迷宫，迷宫可以看成是由 $n * n$ 的格点组成，每个格点只有2种状态，.和#，前者表示可以通行后者表示不能通行。同时当Extense 处在某个格点时，他只能移动到东南西北(或者说上下左右)四个方向之一的相邻格点上，Extense 想要从点A走到点B，问在不走出迷宫的情况下能不能办到。如果起点或者终点有一个不能通行(为#)，则看成无法办到。





# ◎ 作业

## • 4. 算 24 (P189)

- 给出4个小于10个正整数，你可以使用加减乘除4种运算以及括号把这4个数连接起来得到一个表达式。现在的问题是，是否存在一种方式使得得到的表达式的结果等于24。这里加减乘除以及括号的运算结果和运算的优先级跟我们平常的定义一致（这里的除法定义是实数除法）。比如，对于5，5，5，1，我们知道 $5 * (5 - 1 / 5) = 24$ ，因此可以得到24。又比如，对于1，1，4，2，我们怎么都不能得到24。

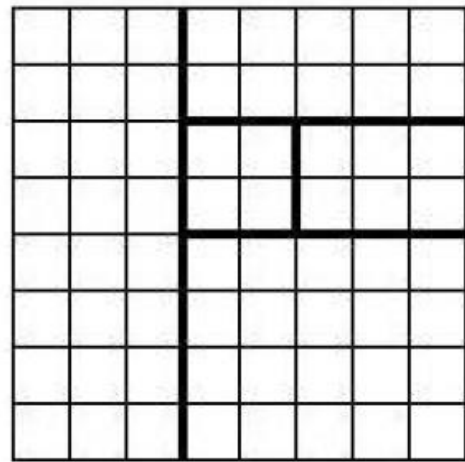




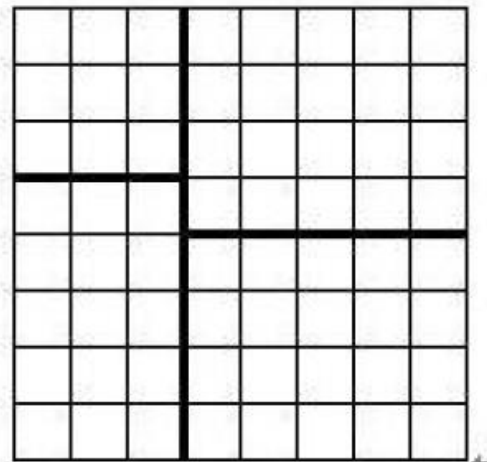
# ◎ 作业

## • 8. 棋盘分割 (P191)

- 将一个  $8 \times 8$  的棋盘进行如下分割：将原棋盘割下一块矩形棋盘并使剩下部分也是矩形，再将剩下的部分继续如此分割，这样割了  $(n-1)$  次后，连同最后剩下的矩形棋盘共有  $n$  块矩形棋盘。(每次切割都只能沿着棋盘格子的边进行)原棋盘上每一格有一个分值，一块矩形棋盘的总分为其所含各格分值之和。现在需要把棋盘按上述规则分割成  $n$  块矩形棋盘，并使各矩形棋盘总分的均方差最小。请编程对给出的棋盘及  $n$ ，求出均方差的最小值。



允许的分割方案



不允许的分割方案