计算机程序设计

Computer Programming

函数

主讲：吴锋

目录
CONTENTS

◎ 函数的作用

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return(0);
}
```

- printf() 体现了函数的重要性：
  - 可以完成特定功能（按指定格式打印字符串）
  - 屏蔽功能的实现细节（如果没有printf函数可用，必须直接跟操作系统甚至输出硬件，如显示屏，打交道）
  - 可以被任意程序反复调用，减少编程工作量
  - 便于模块化、结构化程序设计，简化程序调试
  - 支持任务分工，使大规模软件开发成为可能
  - ……

◎ 函数的作用

- 一般的C程序包含一个main函数及多个其它函数，其中的函数可以是：
  - 库函数：由程序开发环境提供，或从第三方获得，常为.lib文件（需要在开发环境中或使用工具软件查看）；不同的C语言编译系统提供的库函数的数量和功能会有一些不同
  - 用户自定义函数：由用户自己编写，为文本形式（可用记事本等文本编辑软件查看）

- 使用函数的规则
  - 在ANSI C中，与变量类似，函数需要在使用（称为函数调用）前进行定义或声明
  - main() 是特殊函数，无需事先定义或声明，可以调用其它函数，但不能被其它函数调用
  - 其它任何函数在定义或声明后，都可以被一个或多个函数调用任意多次

◎ 函数举例：计算圆的面积

用户自定义函数

主函数

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
int main() {
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0);  //调用math.h中声明的函数
    a=area(r);     //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
```

◎ 函数的组成成分

- 函数名：
  - 函数的名字，应尽量准确地反映函数功能（有命名规则）
  - 同一程序/文件中不能重名
  - 一个工程文件中，有且只能有一个main()函数
- 返回值类型：也称为函数类型
  - 规定了被调用的函数执行完成后返回值的数据类型
  - 不需要返回值时，应定义为void（空类型）
  - 缺省则为int类型

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
int main() {
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0); //调用math.h中声明的函数
    a=area(r);    //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
```

◎ 函数的组成成分

- 形式参数：简称形参
  - 形参表规定了函数调用时传递的数据对象的个数、顺序及数据类型
  - 系统临时分配存储单元，以接收并存放调用时传递进来的数据（地位相当于函数的内部变量）
- 函数返回值：
  - return语句表示被调用函数完成功能并返回，可以返回一个值(按函数名前指定的类型)或不返回值(无论函数名前是否有返回类型)
  - exit()语句表示终止程序运行。main()函数中return和exit语句等价

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
int main() {
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0); //调用math.h中声明的函数
    a=area(r);    //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
```

◎ 函数的组成成分

- 局部变量：
  - 其作用域在函数内部（其它函数不能使用）；
  - 形参虽然在函数首部说明，但作用域同内部变量；
  - 形参可以认为是被初始化了的内部变量，因此二者不能重名
- 执行语句：
  - 完成特定的功能；其中不能嵌套定义函数，但可以调用其它函数，甚至调用自己（递归调用）
- 局部变量声明和执行语句合称函数体

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
int main() {
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0);  //调用math.h中声明的函数
    a=area(r);     //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
```

◎ 函数的组成成分

- 特例：空函数

  返回值类型说明　函数名() { }

  - 如：void null () { }
  - 作用：可正常调用，但函数本身什么都不做，通常用来占位，开发大型程序时常见

- 关于main()函数：
  - 由于main()函数不会被其它函数调用，因此其返回值类型可以随便定义，return语句也可有可无（实际编译器有不同要求）
  - 规范的写法是：

```
int main() {...; return(0);}

void main() {...}
```

◎ 函数的定义与声明

- 函数名必须先定义或声明，然后才能使用

- 函数的定义是告诉编译器，名字所要完成的具体任务（做什么，怎么做）
- 函数的声明是告诉编译器，如何解释和使用名字（做什么）

- 函数的定义是函数的完整代码，包含了函数所需要的的所有成分
  - 函数定义也具有声明的作用。若被调函数定义在主调函数之前，则可以不对被调函数进行声明，因为编译系统已经知道被调函数的所有特征
  - 否则应先给出函数原型声明，然后才可以调用

◎ 函数的定义与声明

- 函数原型声明是对已定义函数或准备定义函数的函数名、函数类型、形参表等信息进行说明，不包含函数体

- 函数原型声明的两种形式（效果相同）：

  函数类型 函数名 (参数类型1，参数类型2，...);

  如： int max (int, int);

  函数类型 函数名 (类型1 形参1，类型2 形参2，...);

  如： int max (int x, int y);

◎ 函数的定义与声明

- 注意区分函数原型声明与函数定义
  - 形式上
    - 原型声明不包括函数体，且以“;”结尾
    - 函数定义包括函数体
  - 作用上
    - 原型声明仅是通知编译系统被调函数的特征，用以验证用户调用函数合法性
    - 函数定义则是整个函数的完整代码

◎ 函数举例：计算圆的面积（函数定义置后）

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
int main() {
    double area(double x);   //函数声明
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0);
    a=area(r);      //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
```

```
#include<stdio.h>
#include<math.h>
#define PI 3.14159
double area(double x);   //外部声明函数
int main() {
    double r,a;
    printf("Input: r=?\n");
    scanf("%lf",&r);
    r=pow(r,2.0);
    a=area(r);      //调用用户自定义函数
    printf("The area is %f\n", a);
    return(0);
}
double area(double x) {
    double y=0;
    y=PI*x;
    return(y);
}
```

函数的定义与声明

- 函数声明可以在主调函数的声明部分，也可以在函数外部。通常集中放在文件的前面，或在头文件 (.h) 中
  - 建议将函数原型声明放在头文件中，并在定义函数的文件中包含该头文件，以便编译器及时发现不匹配的情况
- 在外部声明的函数名，可被之后所有函数调用，无需在函数内再次声明
- 库函数的声明包含在头文件 (*.h) 里，包含了头文件就无须另外写

◎ 函数调用

- 函数调用的一般形式

  函数名(实参表)

  - 例如：r=pow(r, 2.0); a=area(r);
  - 实参，即实际传递给函数的参数（或表达式）
  - 实参表中实参的个数、出现的顺序和实参的类型一般应与函数定义中形参的设置相同
  - 如果函数定义中没有形参，则函数调用时也没有实参，但函数名后的括号不能省
  - 当实参表中有多个实参时，各实参之间要以逗号分开

◎ 函数调用

- 函数调用时，存在流程控制转移和数据传递
- 调用者称为主调函数，被调用者称为被调函数

- 当被调函数存在形参时，主调函数向其传递“实际参数”（简称实参），并将流程控制转移到被调函数，被调函数的形参接收实参后，执行并完成预定的任务

- 被调函数执行完语句后，把流程控制返回主调函数调用它的地方，并通过函数返回值向主调函数返回信息

函数调用举例

- 计算并输出三个电阻的串联和并联值，分别由函数series()和parallel()实现

```
#include<stdio.h>
float series(float a1,float a2,float a3) {
    return(a1+a2+a3);
}
float parallel(float b1,float b2,float b3) {
    float rp,rr;
    rr=1.0/b1+1.0/b2+1.0/b3;
    rp=1.0/rr;
    return(rp);
}
int main() {
    float r1,r2,r3,rs,rp;
    scanf("%f%f%f",&r1,&r2,&r3);
    rs=series(r1,r2,r3);
    printf("The series values is %f\n",rs);
    rp=parallel(r1,r2,r3);
    printf("The parallel values is %f\n",rp);
}
```

◎ 函数调用

- 按函数调用在程序中出现的形式和位置来分，可以有以下3种函数调用方式：

1. 函数调用语句

- 把函数调用单独作为一个语句

  如： printf (“Hello world!”); strcpy(str1,str2);

- 不要求函数带回值（虽然这两个函数实际上是有返回值的），只要求函数完成一定的操作

2. 函数表达式

- 函数调用也是表达式，可以出现在任何表达式中

  如： c=max(a,b); rs=series(r1,r2,r3);

- 此时函数必须返回一个确定的值，以参与运算

◎ 函数调用

3. 函数参数

- 函数调用作为另一函数调用时的实参

  如 printf("%f", series(r1,r2,r3));

  m＝max(a, max(b,c));

- 本质都是函数返回值参与后续运算
  - 函数的返回值才是它的“本职工作”
  - 函数附带的效果，例如显示等，是它的“副作用”

◎ 参数传递

- 函数定义中的形式参数
  - 形参用于定义函数拟接收数据的类型和顺序
  - 函数也可以没有形参，表示函数不需要接收数据
  - 形参只是一种说明，不能初始化（C++支持缺省参数，可以初始化）

- 函数调用时的实际参数
  - 实参可以是常量、变量或表达式，但必须有确定的值
  - 实参的个数、类型应与形参一致，个数不一致会出错，类型不一致会进行隐式转换（随编译器不同，可能发生意想不到的结果）

◎ 参数传递

- C语言参数的传递遵循“值拷贝”原则，并分成两种情况
  - 当实参是常量、变量或表达式时，传递的是数据对象的内容（值），简称传值方式
- 例：

```
#include<stdio.h>
void swap(int x,int y)
{    int t;
     t=x; x=y; y=t;
     printf(“in swap():x=%d y=%d\n”,x,y);
}
int main()
{    int a=3,b=5;
     printf(“Before call swap():a=%d b=%d\n”,a,b);
     swap(a,b);
     printf(“After call swap():a=%d b=%d\n”,a,b);
}
```

运行结果是：

```
Before... :a=3 b=5
in        ...:x=5 y=3
After  ...  :a=3 b=5
```

◎ 参数传递

- C语言参数的传递遵循“值拷贝”原则，并分成两种情况
  - 当实参是常量、变量或表达式时，传递的是数据对象的内容（值），简称传值方式
  - 当实参是数据对象的首地址值时（如数组名），简称传址方式或传地址方式
    - 传递的是地址的值
    - 被调函数通过地址访问（读/写）数据存储空间——具有数据双向传递的功效
- 例：

```
#include<stdio.h>
void add(int b[], int n) {
    int i;
    for(i=0; i<n; i++) b[i]++;
}
int main() {
    int a[5]={1,2,3,4,5},i;
    add(a,5);
    for(i=0; i<5; i++) printf("a[%d]=%d\n",i+1,a[i]);
}
```

运行结果是：
a[5]={2,3,4,5,6}

◎ 参数传递

- 当有多个实参时，实参的计算顺序未严格规定
  - 实参规定是从右向左入栈的
  - 实参的计算未见严格定义，但与入栈顺序一致是最容易实现的

- 要警惕赋值相关运算的副作用发生时间
  - 例如：printf(“%d,%d,%d\n”, i++, i++, i++); 在VC6中调用的是..., i, i, i，在Dev C++中调用的是...i+2, i+1, i

- 应避免使用类似未严格定义的写法

◎ 参数传递

- return语句
  - return语句的作用：结束当前函数的执行，并将返回值传给主调函数。例：

```
int test(int n)
{   if(n>10) return 1;
    return(0);
}
```

注意return语句两种不同的格式，等效

  - 当return的数值类型与函数类型不一致时，会转换成函数类型
  - 函数需要返回值时，若缺少return语句，或return语句未带返回值，则返回一个不确定值（多数编译器会给出警告）
  - C语言的函数不需要返回值时，可以使用void来将函数定义为空类型，此时，函数体中不出现return语句；或使用不带返回值的return语句：return;

◎ 数组作为函数参数

- 一维数组元素作为函数参数
  - 数组元素是单值变量，因此作实参使用时和一般变量没什么区别
- 一维数组名作为函数参数
  - 函数的形参如果是数组，调用时实参要用数组名，此时是传地址调用，形参和实参指向同一块存储空间，相当于对这块空间分别命名
  - 事实上，数组作为函数参数时，无论函数调用前后，都不会为形参数组分配另外的存储空间，形参数组只接收实参数组的首地址（即指针）
  - 实参数组名是个常数，不可更改；形参数组名是个指针，其值可以更改
  - 一维数组作形参时，一般不定义数组的大小，即使定义，也没有实际意义。传递的仅是首地址，并不关心有多少元素。数组元素个数一般另用参数传递
  - 实参也可以不用数组名，而是某段内存的起始地址。至于内容，要由程序员掌控

◎ 数组作为函数参数（例）

- 例：数组分段交换，即将一个长度为n的数组a分为两段，前k个元素一段，后n-k个元素一段，要求编写函数实现两段交换，例如：

  a={1,2,3,4,5,6,7,8,9}, k=3交换后变成

  a={4,5,6,7,8,9,1,2,3}

- 最直接的思路是通过一个临时数组暂存和处理

◎ 数组作为函数参数（例）

```
#include <stdio.h>
void array_swap(int a[], int n, int k) {
    int t[30],i;
    for (i=0; i<k; i++) t[i]=a[i];
    for (i=k; i<n; i++) a[i-k]=a[i];
    for (i=0; i<k; i++) a[n-k+i]=t[k];
}
void main() {
    int a[]={1,2,3,4,5,6,7,8,9};
    int i, n=9, k=3;
    for (i=0; i<n; i++) printf("%3d",a[i]);
    printf("\n");
    array_swap(a,n,k);
    for (i=0; i<n; i++) printf("%3d",a[i]);
    printf("\n");
}
```

◎ 数组作为函数参数

- 二维数组元素作为函数参数
  - 数组元素是单值变量，因此作实参使用时和一般变量没什么区别

- 二维数组名作为函数参数
  - 行优先排列，有序地占用一片连续的内存区域
  - 形参定义成二维数组时，必须指定第二维的大小，编译器才知道如何由数组下标计算元素存储位置
  - 第一维行的大小可以根据需要选择传递，以告知数组大小
  - 因为是地址传递，原则上并不要求形参数组与实参数组的结构一样，被调用函数按照形参指定的格式理解数据。但不一致时很难理解，极少使用

◎ 数组作为函数参数（例）

- 例：矩阵乘法

```
#include<stdio.h>
void matpro (int a[][3], int b[][2], int c[][2], int m, int n, int p) {
    int i,j,k,s;
    for (i=0; i<m; i++)
        for (j=0; j<p; j++){
            for (k=s=0; k<n; k++) s+=a[i][k]*b[k][j];
            c[i][j]=s;
        }
}
void main() {
    static int a[2][3]=..., b[3][2]=..., c[2][2];
    ...;
    matpro(a,b,c,2,3,2);
    ...
}
```

◎ 数组作为函数参数

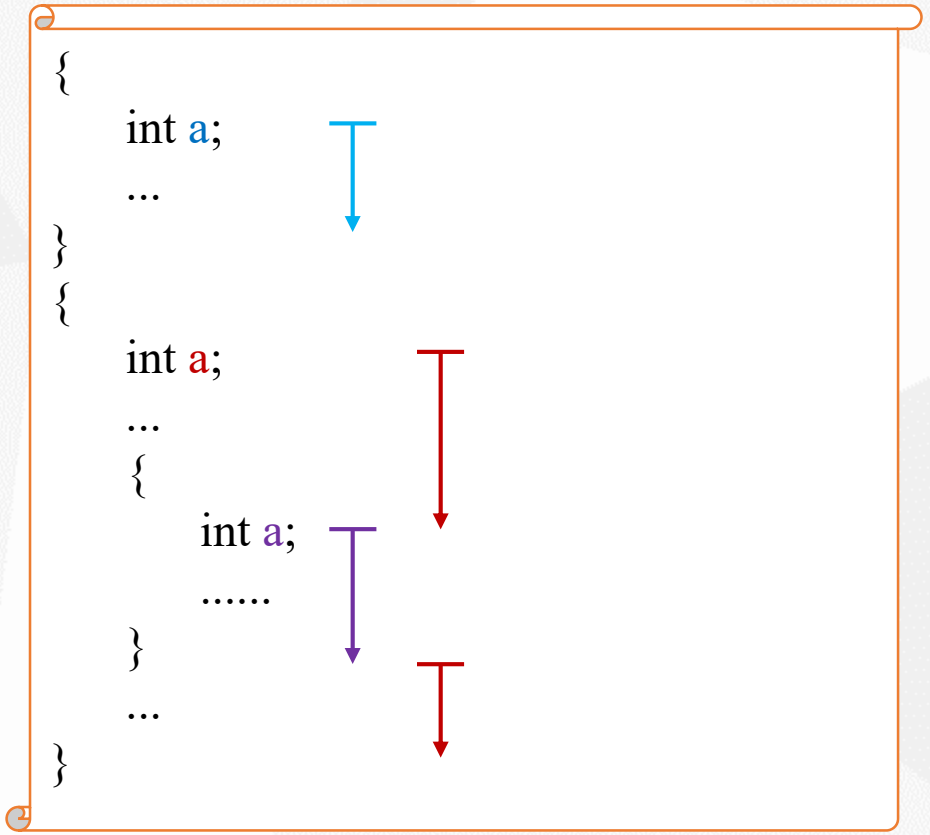
- 多维数组作为函数参数
  - 与二维情况类似
  - 多维数组名作为函数参数时，形参定义需要指定第二维及之后所有维度的大小，以便寻址

局部变量与全局变量

- 变量声明的位置有三种
  - 在函数的开头，称为局部变量
  - 在语句（特别是循环语句或复合语句）中，称为局部变量
  - 在函数的外面，称为全局变量（注意与外部变量不同）
- 作用域
  - 即该变量名字可以被使用的范围
  - 局部变量限制在本程序块内
    - 在函数开头声明的，作用域直到本函数结束
    - 在语句中声明的，作用域直到本语句结束
  - 全局变量限制在本源代码文件内

局部变量与全局变量

- 同一个程序块内，不允许出现重名变量
- 嵌套的程序块出现重名局部变量时，使用本层次或最接近本层次的外层程序块的声明
- 例：



局部变量与全局变量

- 全局变量的使用应适度
  - 是全局可访问的公共数据
  - 节省函数参数的传递
  - 破坏了函数/类的封装
  - 无论数据是否有用，始终占据存储单元
  - 多线程程序可能导致冲突和死锁

◎ 变量的存储类别

- 自动的 auto
  - 例，auto int a;
  - 局部变量缺省都是自动的
  - 在函数被调用时自动分配存储空间，在函数退出时自动释放
- 静态的 static
  - 例，static int a;
  - 全局变量都是静态的。局部变量可以声明为静态的
  - 在程序载入时分配且独占存储空间，在程序退出时释放
  - 对全局变量声明static，其含义是“内部的”，限制该变量只能被本文件使用
- 寄存器的 register
  - 例，register int a;
  - 变量优先存储在寄存器中
- 外部的 extern
  - 例，extern int a;
  - 声明变量a在其它文件中，链接时进行拼装

◎ 静态变量

- 静态变量存储在静态数据区（关键字 static）
  - 在程序运行时只初始化一次。
  - 作用域是局部于函数的，函数外不可见；如果是在函数外定义的静态变量，则局部于文件，文件外不可见
  - 生命期是全局的（程序级），每次函数调用，保留上次函数调用后的值，不随着函数的退出而销毁

变量的存储类别

- 静态局部变量的例子

```
#include <stdio.h>
int main() {
    int fac(int n);
    int i;
    for (i=1; i<=5; i++)
        printf("%d!=%d\n", i, fac(i);
    return 0;
}
int fac(int n) {
    static int f=1;
    f=f*n;
    return(f);
}
```

程序运行结果：

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

◎ 寄存器变量

- 寄存器变量存储于寄存器中 （关键字 register）
  - 在函数中频繁使用的局部变量可以定义为寄存器变量(用register修饰)，只要还有寄存器可用，编译器会尽可能将其放入寄存器中，提高执行速度。
  - 关键字register对编译器来说只是一个建议，有些编译器可能忽略该建议，而是使用寄存器分配算法找出最合适的候选放到机器可用的寄存器中。

```
for (register int i = 0; i < s2; i++){
        ia[i] = i;
}
```

◎ 外部变量

- 外部变量是在其它文件中定义的全局变量
  - 关键字 extern 声明该变量将在其它文件中定义，声明不会为变量分配内存，可以多次出现
  - 在一个程序中，一个全局变量只能有一个定义，重复定义的话，在链接过程中会报错

```
// 文件：file1.cpp
int i;
void func1( ) {
      i = 50; printf("func1:i(static)=%d\n", i);
}
// 文件：file2.cpp
void func2() {
      int i =15; printf("func2:i(auto)=%d\n", i);
      if (i) { extern int i; printf("func2:i(extern)=%d\n", i); }
}
```

◎ 变量的存储

- 静态存储（生命期：程序级）
  - 函数外的外部变量
  - 函数内的静态变量（static）
- 栈存储（生命期：函数级）
  - 函数内部定义的（自动，auto）变量
- 寄存器存储（生命期：区块级）
  - 寄存器变量（register）

◎ 排序算法（例）

对n个数进行排序（以从小到大为例）

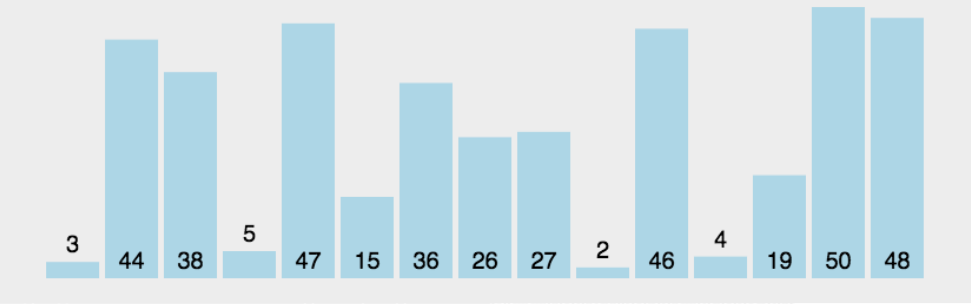
- 交换排序法
  - 第i轮将第i个数与后面所有数依次比较，不满足小于等于关系时交换（或仅与最小的数交换）。每轮排好第i小的数
- 冒泡排序法
  - 每轮对相邻的两个数进行比较，不满足小于等于关系时交换。第i轮排好第i大的数
- 选择排序法
  - 第i轮将第i个数与后面所有数依次比较，若第j个数不满足小于等于关系时，将第j个数继续与后面的所有数比较，直至所有数比较完成，得到最小的数，与i交换。每轮排好第i小的数
- 插入排序法
  - 前m个数已排好序，将第m+1个数插入到合适位置。
- 以上算法复杂度均为$O(n^2)$

◎ 排序算法（例）

- 交换排序法
  - 第i轮将第i个数与后面所有数依次比较，不满足小于等于关系时交换（或仅与最小的数交换）。每轮排好第i小的数

```
void swapSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (arr[i] > arr[j]) {
                temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }
}
```

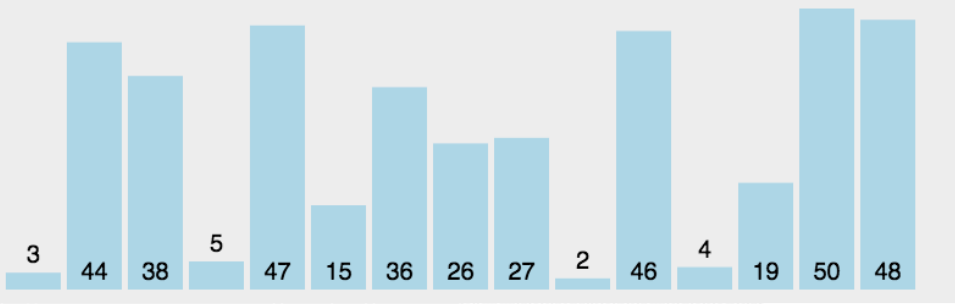
◎ 排序算法（例）



- 冒泡排序法
  - 每轮对相邻的两个数进行比较，不满足小于等于关系时交换。
    第i轮排好第i大的数

```
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++)
            for (j = 0; j < n -1 - i; j++)
                    if (arr[j] > arr[j+1]) {
                            temp = arr[j];
                            arr[j] = arr[j+1];
                            arr[j+1] = temp;
                    }
}
```

◎ 排序算法（例）



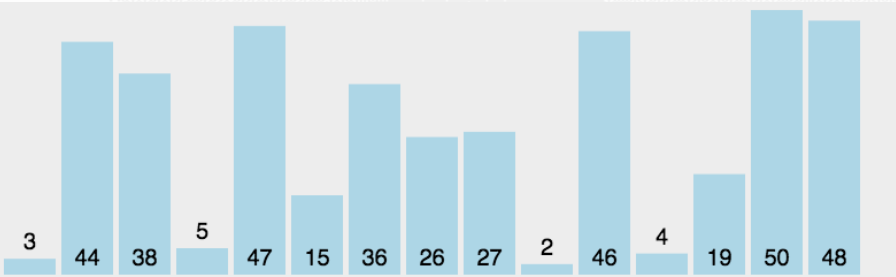
- 选择排序法
  - 第i轮将第i个数与后面所有数依次比较，若第j个数不满足小于等于关系时，将第j个数继续与后面的所有数比较，直至所有数比较完成，得到最小的数，与i交换。每轮排好第i小的数

```
void selectionSort(int arr[], int n) {
    int i, j, temp, minIndex;
    for (i = 0; i < n - 1; i++) {
            for (minIndex=i, j = i + 1; j < n; j++)
                    if (arr[j] < arr[minIndex])
                            minIndex = j;
            temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
 }
```

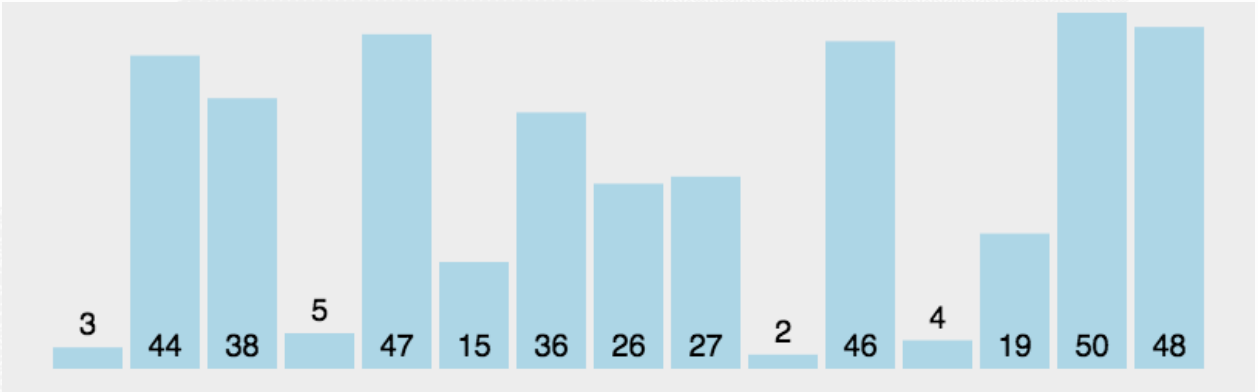
◎ 排序算法（例）

- 插入排序法
  - 前m个数已排好序，将第m+1个数插入到合适位置。

```
void insertionSort(int arr[], int n) {
    int i, preIdx, current;
    for (i = 1; i < n; i++) {
            preIndex = i - 1; current = arr[i];
             while(preIdx >= 0 && arr[preIdx] > current) {
                        arr[preIdx + 1] = arr[preIdx];
                        preIdx--;
            }
            arr[preIdx + 1] = current;
     }
}
```



◎ 排序算法（例）



- 快速排序法
  - 分治法
  - 以第1个数(设为a)为基准，将小于等于a的数挪到数组前半段，将大于等于a的数挪到数组后半段
    - 更好的方法是取第一个元素、中间元素、最后一个元素的中间值
  - 对前半段和后半段两个数组分别递归执行快速排序
  - 算法复杂度为O(n log n)

```
#include <stdlib.h>
int compare(const void * p1, const void *p2) {
    return *(int *)p1 - *(int *)p2;
}
int a[10] = { 2, 4, 1, 5, 5, 3, 7, 4, 1, 5};
qsort(a, 10, sizeof(int), compare);
```

二分法查找算法（例）

在一个已从小到大排序的数组中，查找特定key

- 分治法
  - 双指针，low指向数组开始，high指向数组末尾
  - 计算mid指向low和high的中间
  - 若mid指向的数等于key，则找到
  - 否则根据mid指向的数大于key或小于key，修改low或high，继续迭代查找

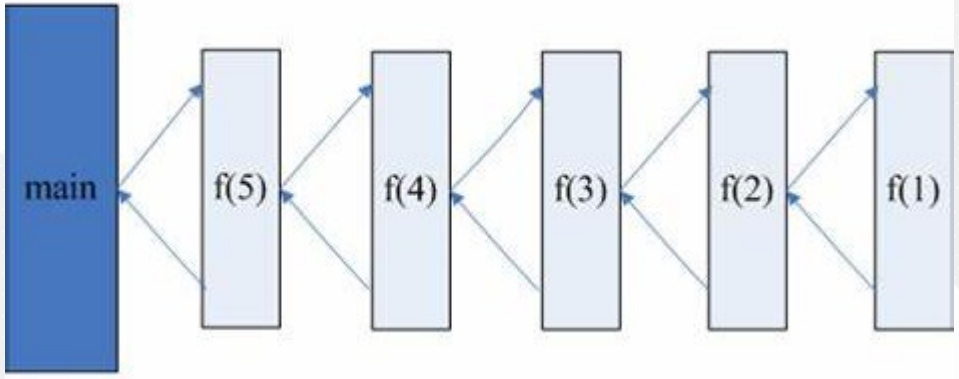
```
#include <stdio.h>
void main() {
    int a[]={5,13,19,21,37,56,64,75,80,88,92};
    int n, key, low, mid, high;
    n=sizeof(a)/sizeof(a[0]);
    scanf("%d", &key);
    low=0; high=n-1;
    while (low<high) {
        mid=(low+high)/2;
        if (key==a[mid]) {
            printf("a[%d]=%d"\n, mid, key);
            break;
        }
        else if (key>a[mid]) low=mid+1;
        else high=mid-1;
    }
    printf("not found!\n");
}
```

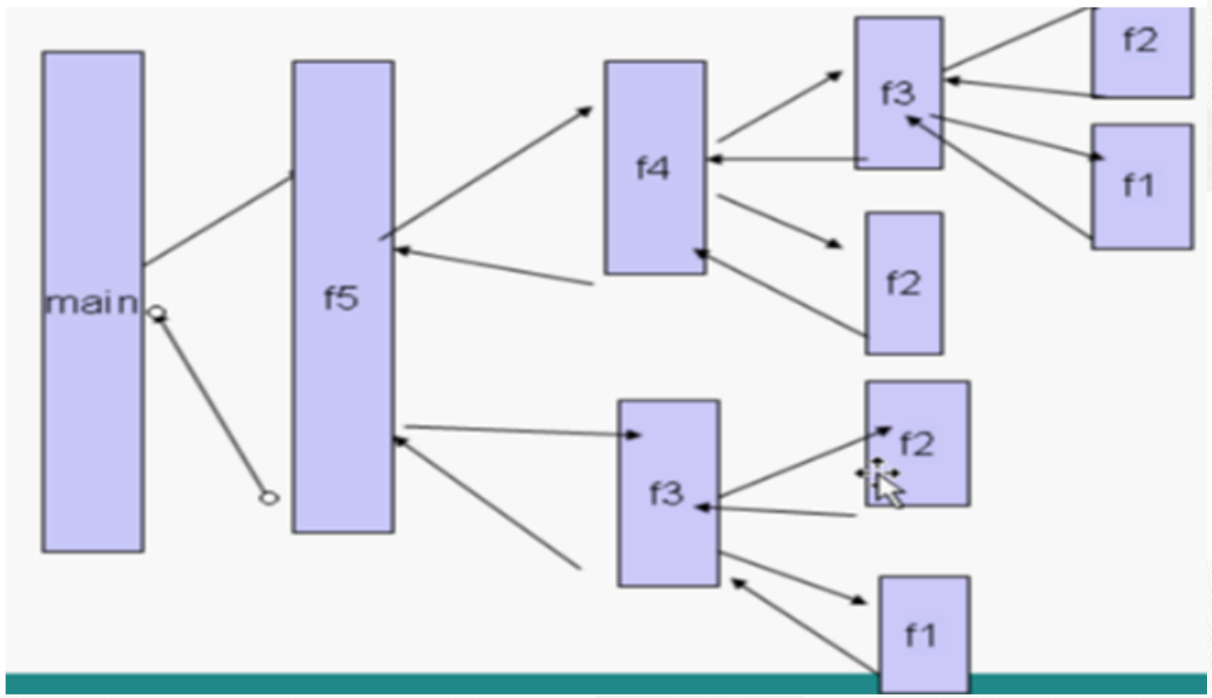
◎ 递归的定义

- 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

- 找出递归的求解策略（递推公式）；
- 找到递归的终止条件（出口条件）；
- 先设计出递归函数原型，然后在函数体中进行递归调用。





◎ 递归举例：阶乘

- 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？

◎ 递归举例：阶乘

- 求阶乘的递推程序：0!=1, 1!= 0!*1 = 1, 2! = 1! *2 = 2

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

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

◎ 递归举例：阶乘

- 求阶乘的递归程序：n! = (n-1)! *n，　0!=1

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

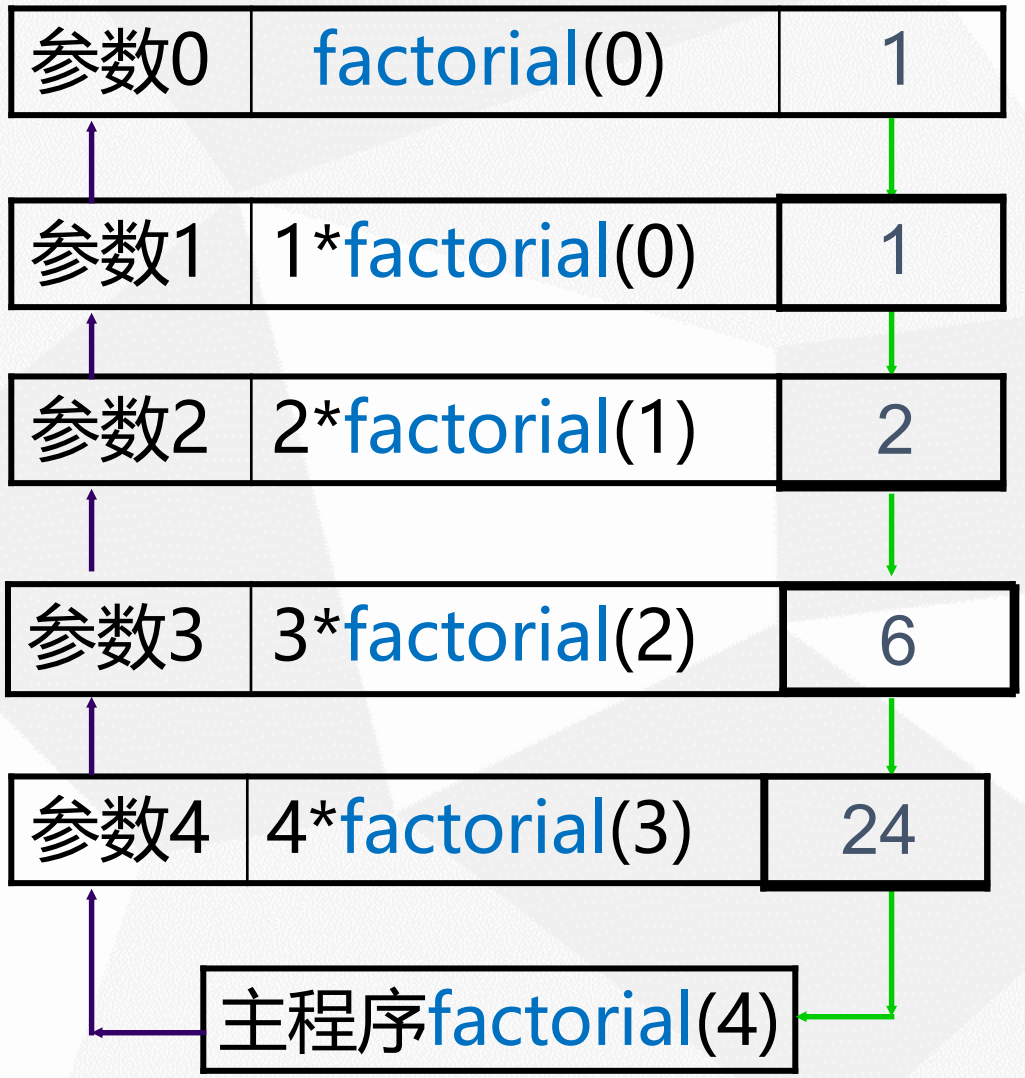
→ 函数原型

出口条件

→ 递推公式

◎ 递归举例：阶乘

```
int  factorial (int n)
{
    // 递归返回
    if (n==0) return 1;
    // 递归策略
    return n*factorial(n-1);
}
```



◎ 递归举例：放苹果

- 问题描述
  - 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。

◎ 递归举例：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：if (d > a) f(a, d) = f(a, a)
  - 当 d <= a 时，不同的放法可以分成两类：
    - 有盘子空着，假定该情况下的放法数目为 g(a, d)
    - 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)

◎ 递归举例：放苹果

- 解题思想：先对a作讨论
  - 当 d <= a 时，不同的放法可以分成两类：
    - 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)
    - 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)

◎ 递归举例：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);
```

◎ 递归举例：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。

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

◎ 递归举例：放苹果

- 具体实现

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

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

◎ 递归举例：神奇口袋

- 问题描述：给出小于40 的正整数$a_1, a_2, \ldots\ldots, a_n$，问凑出和为40 有多少种方法。
- 解题思路
  - 设 $f(a_1, a_2, \ldots\ldots, a_n, 40)$ 为用$a_1, a_2, \ldots\ldots, a_n$ 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：

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

◎ 递归举例：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……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
```

◎ 递归举例：神奇口袋

- 解题思路：出口条件
  1. 如果sum == 0,则应该返回 1
     - 一个数都不取，也是一种取法
  2. 如果sum < 0 则应该返回 0
     - 没法凑出负数
  3. 如果 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？

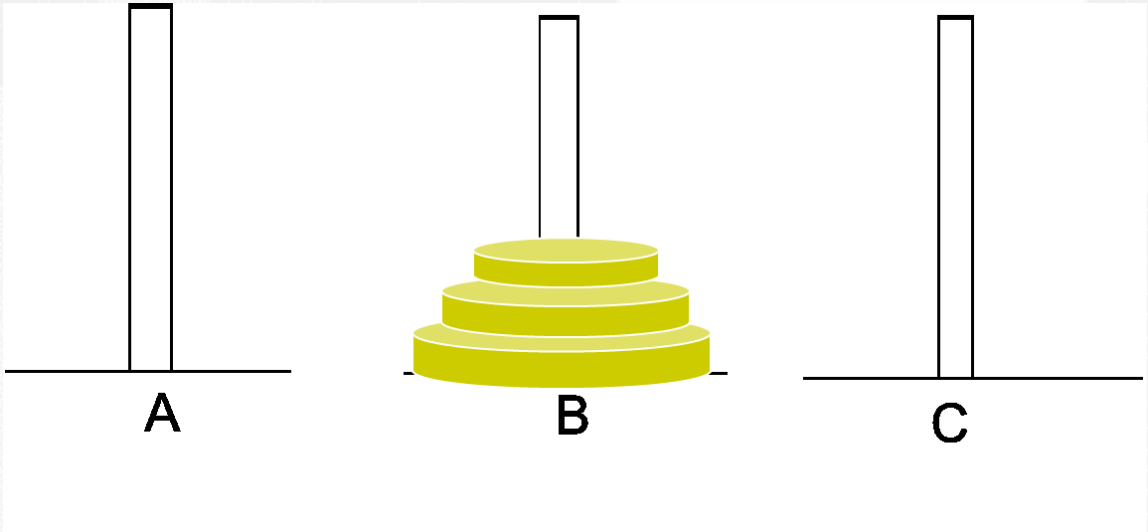
◎ 递归举例：神奇口袋

- 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);
}
```

◎ 递归举例：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。



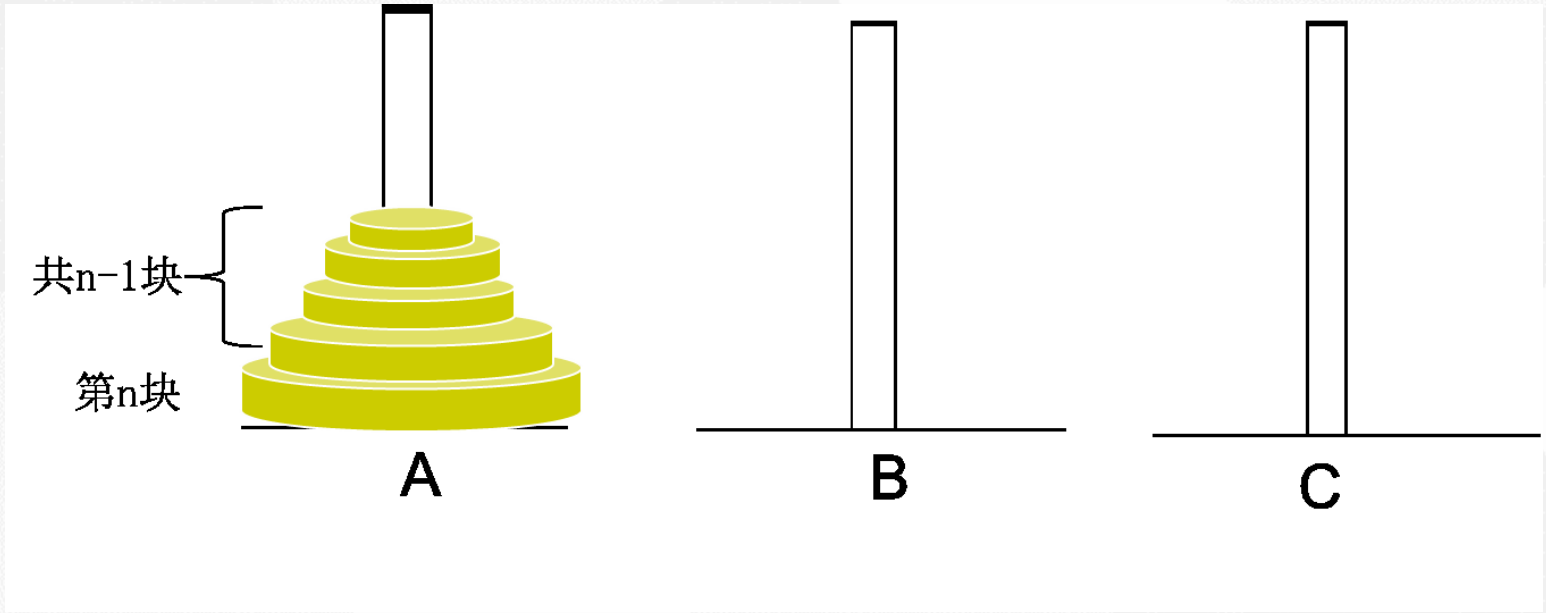
◎ 递归举例：汉诺塔

- 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

◎ 递归举例：汉诺塔

- 问题2：如何应用递归求解？
  - 关键点一：递归的策略



◎ 递归举例：汉诺塔

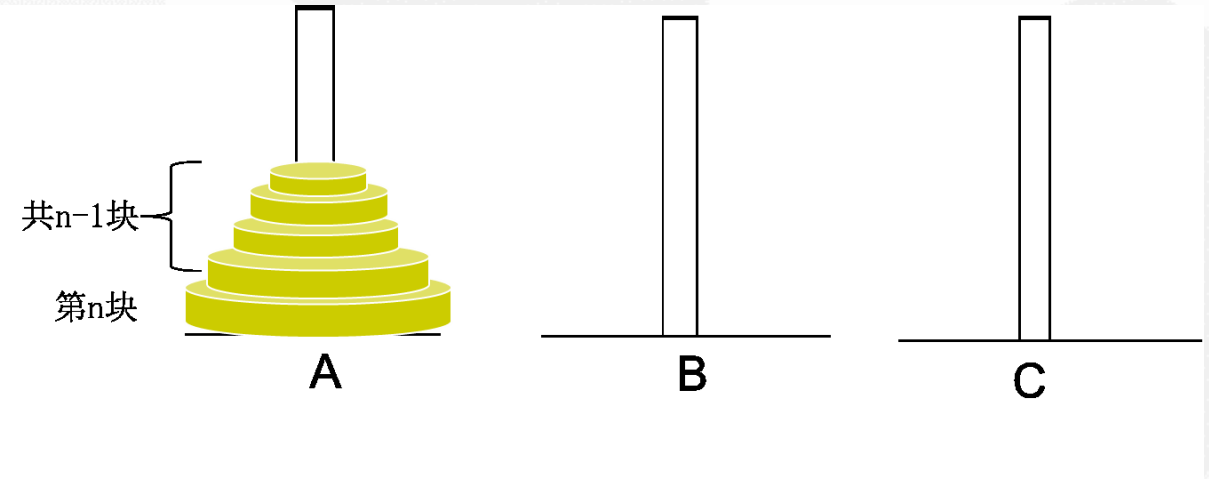
- 问题2：如何应用递归求解？
  - 关键点一：递归的策略

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；

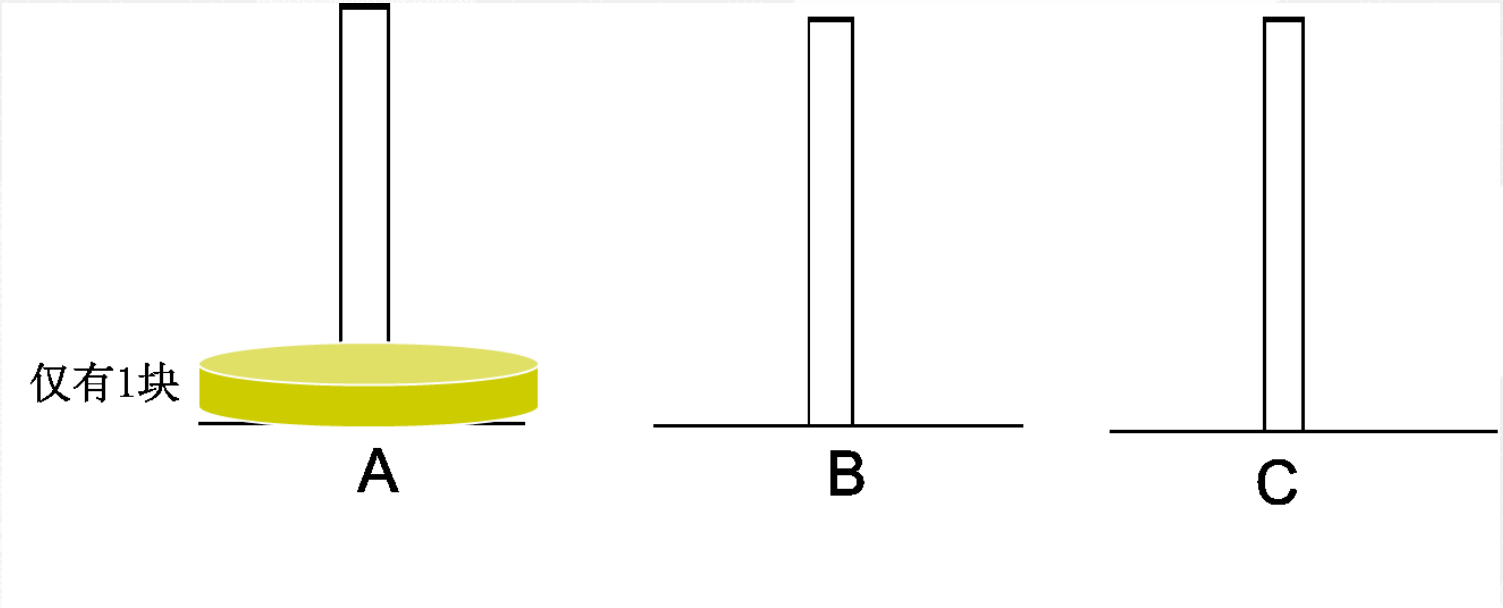
③ 将C柱上的n-1个盘，以A为中转，移动到B柱。



◎ 递归举例：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->B



◎ 递归举例：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

◎ 递归举例：汉诺塔

- 具体实现

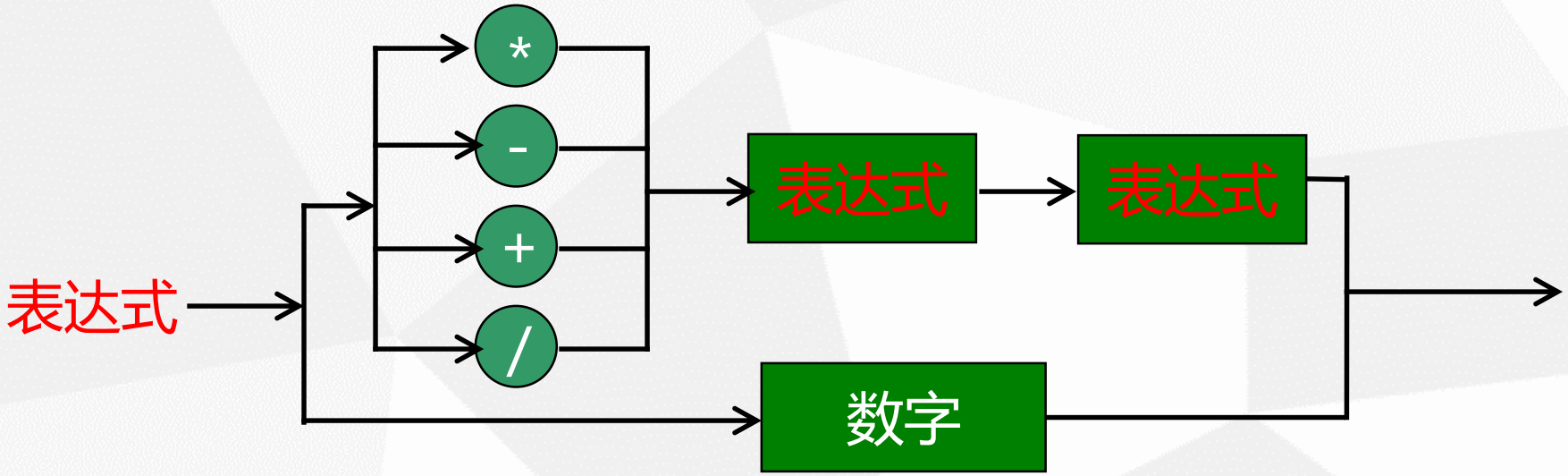
```
#include <stdio.h>
// 将 n 个盘子从 a 柱移动到b柱，用c柱做中转
void Hanoi(int n, char a, char b, char c) {
        // 递归的结束条件：只有1个盘子，直接从a柱移到b柱
        if (n == 1) { printf("%c->%c\n", a, b); return; }

        // ① 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);
        // ② 将一个盘子从a移动到b
        printf("%c->%c\n", a, b);
        // ③ 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);
}
void main() {
        int N;
        scanf("%d", &N);
        Hanoi(N, 'A', 'B', 'C');
}
```

◎ 递归举例：逆波兰表达式

- 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。
- 要求求解逆波兰表达式的值，其中运算符包括 + - * / 。
  - 输入样例： * + 11.0 12.0 + 24.0 35.0
  - 输出样例： 1357.000000

◎ 递归举例：逆波兰表达式

- 逆波兰表达式的递归定义



◎ 递归举例：逆波兰表达式

- 具体实现

```
#include <stdio.h>
#include<string.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
        }
}
void main() {
        double ans;
        ans = exp();
        printf("%f", ans);
}
```

◎ 递归举例：逆波兰表达式

- 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。

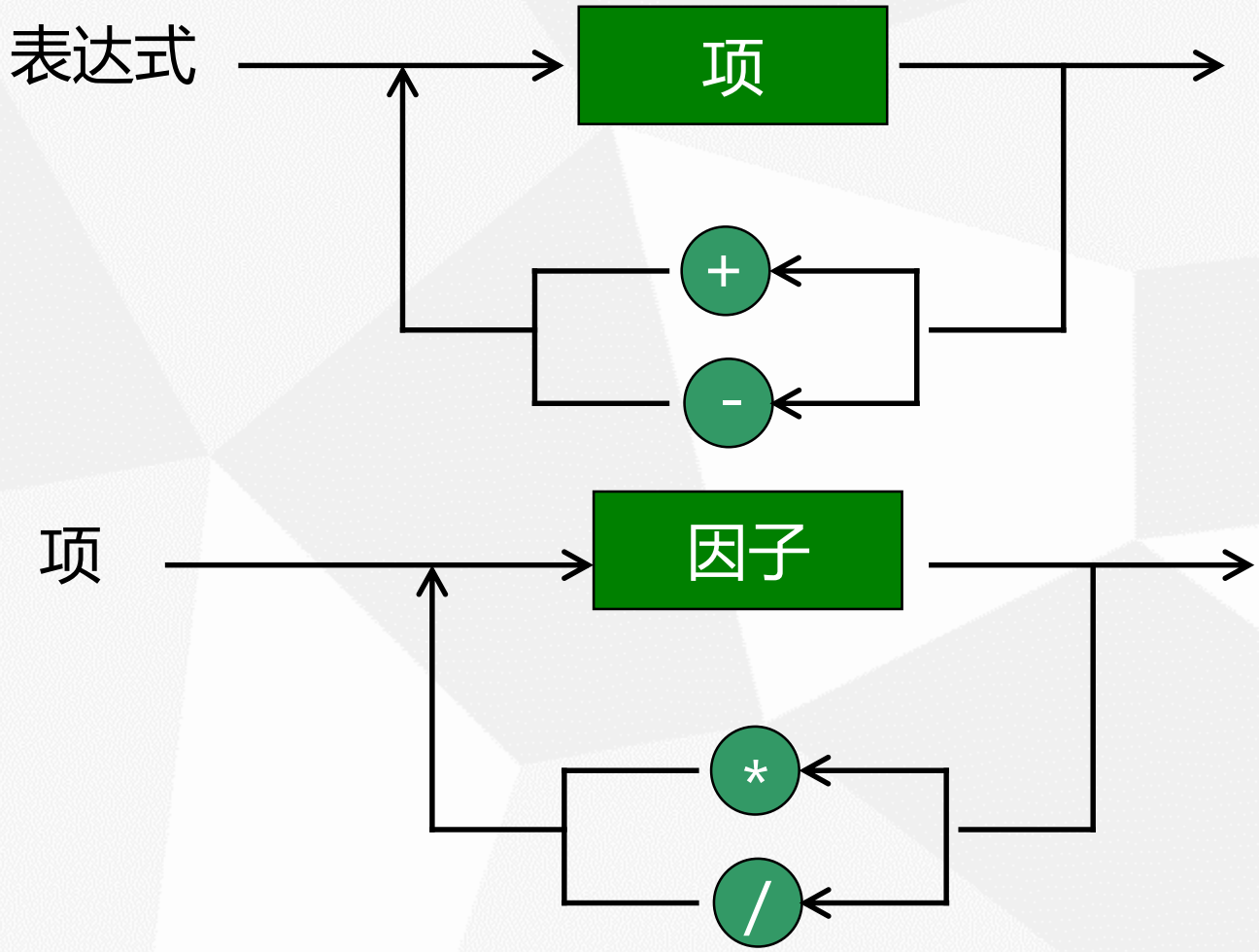
```
#include <stdio.h>
#include <string.h>
void exp() {
    char a[10];
    scanf("%s", a);
    switch (a[0]) {
        case '+':  printf("("); exp(); printf(")"); printf("+");
                   printf("("); exp(); printf(")"); break;
        case '-':  printf("("); exp(); printf(")"); printf("-");
                   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;
    }
}
void main() { exp(); }
```

◎ 递归举例：逆波兰表达式

- 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。

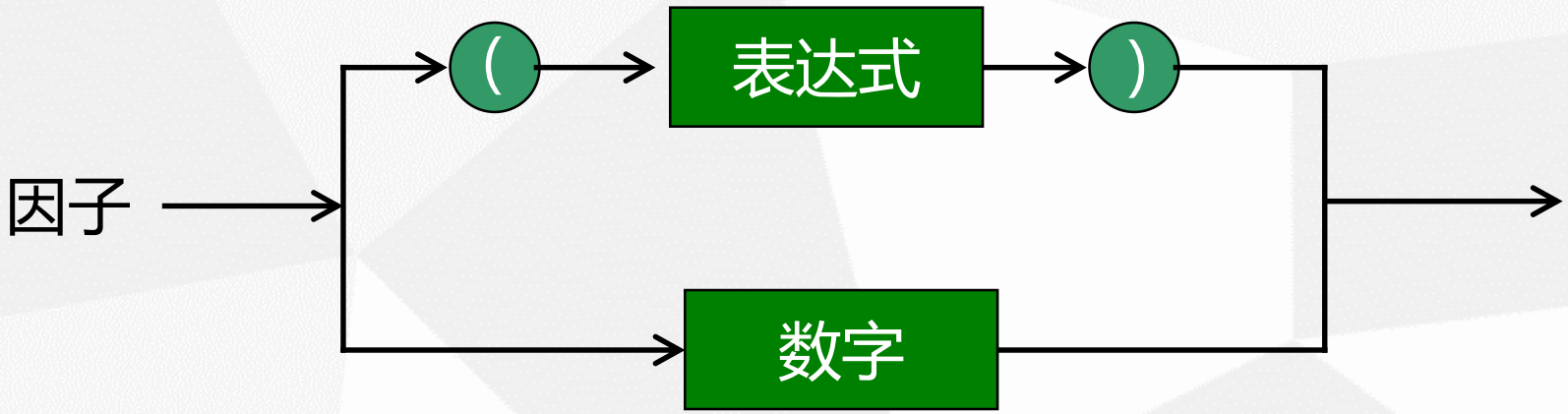
◎ 递归举例：逆波兰表达式

- 常规表达式的递归定义



◎ 递归举例：逆波兰表达式

- 常规表达式的递归定义



◎ 递归举例：逆波兰表达式

- 具体实现

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

double factor_value();
double term_value();
double expression_value();

char curr_char;

void main() {
        curr_char = getchar();
        double result = expression_value();
        printf("The result is %f\n", result);
}
```

◎ 递归举例：逆波兰表达式

- 具体实现

```
// 求一个表达式的值
double expression_value()  {
        double result = term_value(); // 求第一项的值
        int more = 1;
        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 = 0;
        }
        return result;
}
```

◎ 递归举例：逆波兰表达式

- 具体实现

```
//求一个项的值
double term_value()  {
        double result = factor_value(); //求第一个因子的值
        int more = 1;
        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 = 0;
        }
        return result;
}
```

◎ 递归举例：逆波兰表达式

- 具体实现

```
//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 {
                char num[64]; int i = 0;
                while (isdigit(c) || c == '.') {
                        num[i++] = c;
                        c = curr_char = getchar();
                }
                result = atof(num);
        }
        return result;
}
```

◎ 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。

◎ 动态规划的思想

- 例：Fibonacci（斐波那契）数列

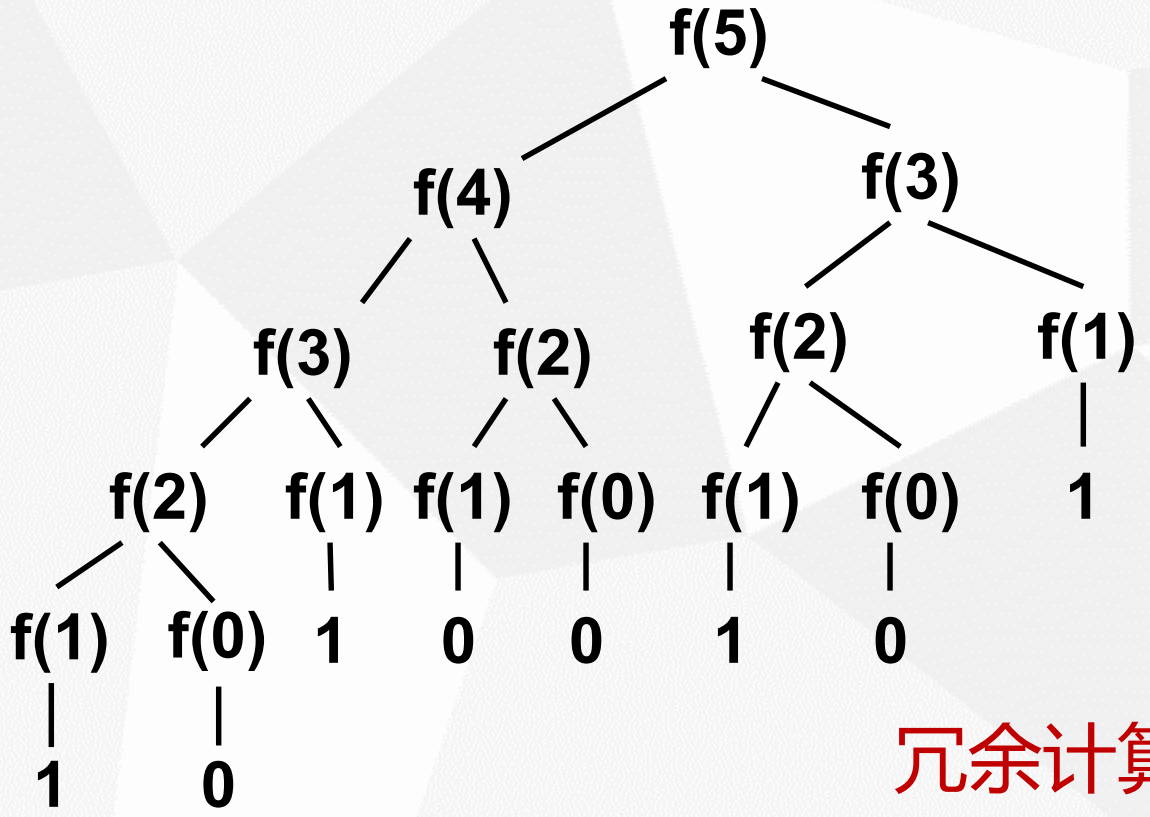
$$\begin{cases} F_1 = 1 & (n = 1) \\ F_2 = 1 & (n = 2) \\ F_n = F_{n-1} + F_{n-2} & (n \geq 3) \end{cases}$$

- 求 Fibonacci数列的第n项

```
int f(int n) {
    if (n==0 || n==1) return n; // 出口条件
    return f(n-1)+f(n-2); // 递推公式
}
```

◎ 动态规划的思想

- 递归存在冗余计算
  - 计算过程中存在冗余计算，为了除去冗余计算可以从已知条件开始计算，并记录计算过程中的中间结果。



◎ 动态规划的思想

- 去除冗余的基本思路：用空间换时间 -> 动态规划

```
int f[n+1];
f[1] = f[2] = 1;
for (int i=3; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
printf("%d\n", f[n]);
```

◎ 动态规划举例：爬楼梯

有n(0<n<=1000)级台阶，每次可以上一级或两级，问有几种上法？

- 递归法/动态规划法
  - 最后一步可以上一级或两级 f(n)=f(n-1)+f(n-2)

```
#include <stdio.h>
int steps(int n) {
    if (n==1) return(1);
    else if (n==0) return(1);
    else return(steps(n-1)+steps(n-2));
}
void main() {
    int n;
    scanf("%d", &n);
    printf("%d ways.\n", steps(n));
}
```

```
#include <stdio.h>
int main() {
    int n, i, f0=1, f1=1;
    scanf("%d", &n);
    for (i=2; i<=n; i++) {
        f1=f0+f1; // i 级台阶的上法
        f0=f1-f0;  // i-1级台阶的上法
    }
    printf("%d ways.\n", f1);
    return 0;
}
```

◎动态规划举例：数字三角形

- 问题描述
  - 在下面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大。路径上的每一步都只能往左下或右下走。
  - 只需要求出这个最大和即可，不必给出具体路径。三角形的行数大于1小于等于100，数字为 0 – 99。

```
          7
        3   8
      8   1   0
    2   7   4   4
  4   5   2   6   5
```

◎动态规划举例：数字三角形

- 解题思路：
  - 以 D(r, j) 表示第r行第j个数字，以 MaxSum(r, j) 代表从第r行的第j个数字到底边的各条路径中，数字之和最大的那条路径的数字之和，则本题是要求 MaxSum (0, 0) 。
  - 递推公式：从某个 D(r, j) 出发，显然下一步只能走 D(r+1, j) 或 D(r+1, j+1)，所以，对于N行的三角形：

```
if (r == N-1)
        MaxSum(r, j) = D(N-1, j)
else
        MaxSum(r, j) = D(r, j) +
                max{ MaxSum(r + 1, j),  MaxSum(r+1, j+1) }
```

◎动态规划举例：数字三角形

- 数字三角形的递归程序：

```
int triangle[MAX][MAX];  // 存储数字三角形
int n; // 数字三角形的总行数

int longestPath(int i, int j) {
    if (i == n) return 0; // 出口条件
    int x = longestPath(i + 1, j);
    int y = longestPath(i + 1, j + 1);
    if (x < y) x = y; // 取最大值
    return x + triangle[i][j]; // 递推公式
}
```

```
      7
    3   8
  8   1   0
2   7   4   4
4   5   2   6   5
```

程序运行超时：存在大量重复计算。

- 如果采用递规的方法，枚举每条路径，存在大量重复计算。
- 其时间复杂度为2^n，当 n≥100，肯定超时。

◎动态规划举例：数字三角形

- 改进思想：
  - 从下往上计算，对于每一点，只需要保留从下面来的路径中和最大的路径的和即可。
  - 因为在它上面的点只关心到达它的最大路径和，不关心它从那条路经上来的。
- 解法1：
  - 如果每算出一个 MaxSum(r, j) 就保存起来，则可以用 **$O(n^2)$** 时间完成计算。因为三角形的数字总数是 **n(n+1)/2** 。
  - 此时需要的存储空间是：

```
int D[100][100]; // 用于存储三角形中的数字
int aMaxSum[100][100]; // 用于存储每个MaxSum(r, j)
```

◎动态规划举例：数字三角形

```
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];  int N;
int aMaxSum[MAX_NUM + 10][MAX_NUM + 10];
int longestPath() {
        for (int j = 1; j <= N; j++)
                aMaxSum[N][j] = D[N][j];
        for (int i = N ; i > 1; i--)
                for (int j = 1; j < i; j++) {
                        if (aMaxSum[i][j] > aMaxSum[i][j+1])
                                aMaxSum[i-1][j] = aMaxSum[i][j] + D[i-1][j];
                        else
                                aMaxSum[i-1][j] = aMaxSum[i][j+1] + D[i-1][j];
                }
        return aMaxSum[1][1]);
}
```

◎动态规划举例：数字三角形

- 解法2:
  - 没必要用二维 Sum 数组存储每一个 MaxSum(r, j), 只要从底层一行行向上递推，那么只要一维数组 Sum[100] 即可,即只要存储一行的 MaxSum 值就可以。
  - 比解法一改进之处在于：节省空间，时间复杂度不变。

◎动态规划举例：数字三角形

```
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10]; int N;
int aMaxSum[MAX_NUM + 10];
int longestPath() {
        for (int j = 1; j <= N; j++)
                aMaxSum[j] = D[N][j];
        for (int i = N ; i > 1; i--)
                for (int j = 1; j < i; j++) {
                        if (aMaxSum[j] > aMaxSum[j+1])
                                aMaxSum[j] = aMaxSum[j] + D[i-1][j];
                        else
                                aMaxSum[j] = aMaxSum[j+1] + D[i-1][j];
                }
        return aMaxSum[1];
}
```

◎ 递归到动态规化的转化

- 递归到动态规化的转化的一般方法：
  - 原来递归函数有几个参数，就定义一个几维的数组，用于存储中间结果。
  - 数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值。
  - 这样就可以从边界开始，逐步填充数组，相当于计算递归函数值的逆过程。

上机作业

1. 编写程序，读取一个整型数组（元素不超过10000），一边读取一边用插入排序法进行非递减排序。
   - 读取数组、排序、输出数组，用三个函数来实现。
   - 插入排序：假设前t个数据已排好序，将第t+1个数据插入到前面适当位置，使得前t+1个数据也是有序的。
2. 编写程序，输入一个数字字符串s，将它转换为科学计数法输出。
   - 数字串s仅包含数字和小数点，小数点位于最后时，小数点可有可无；小数点前只有0时，0可有可无。
   - 科学计数法应保持原数字的有效精度，例如 10.0 = 1.00E+1；科学计数法的指数部分用E表示，不缺省；E后用一位表示指数的符号位，不缺省。
   - 要求编写函数，通过字符串的操作（不能直接使用printf的格式化输出）来完成转换。

上机作业

3. 编写程序，输入一段文本，暂存在内存中，按照下列规则处理后输出：

① 如果读入的字符是空格、数字 、逗号、句号 或英文字母，则暂存在内存中；

② 如果输入一个字符 '@'，则表示删除在 '@' 之前输入的一个字符；

③ 如果输入一个字符 '$'，则表示删除当前输入的一行字符；

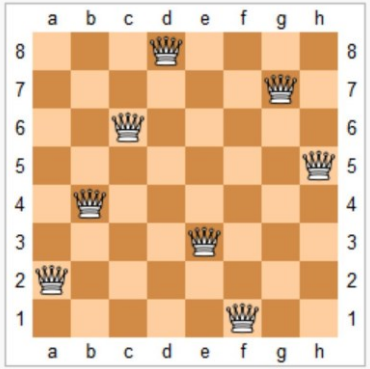
④ 如果输入一个字符 '^'，则表示需要把在 '^' 之前输入的一个英文单词首字母转换成大写字母；

⑤ 如果输入一个字符 '#'，则表示输入结束，将内存的内容输出。

⑥ 如果输入除上述字符之外的其他字符，则直接忽略。

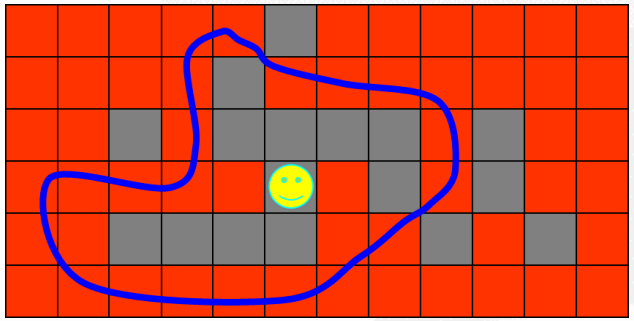
上机作业

4. 用递归法求解N皇后问题（N<=8)。



One solution to the eight queens puzzle

5. 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。



4. 算 24：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。