



# 《计算概论A》课程 程序设计部分

## 函数的递归调用 (2)

李 戈

北京大学 信息科学技术学院 软件研究所

[lige@sei.pku.edu.cn](mailto:lige@sei.pku.edu.cn)



北京大学



## 方法总结之一

### ■ 如何利用递归解决问题？

- ◆ 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，进而找到第 $i$ 次做与第 $i-1$ 次做之间的关系；

- ◆ 确认哪一次做时不需要“再重复”（通常是 $n=1$ 的时候），而且能够给前一次返回结果；



北京大学



## 方法总结之二

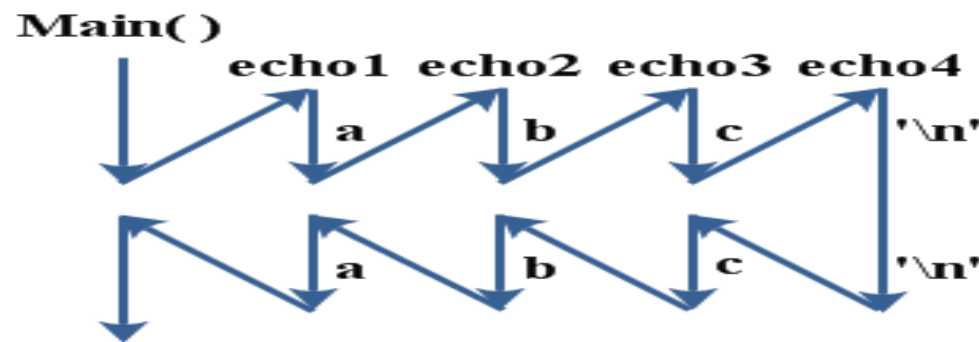
### ■ 如何利用递归解决问题？

#### ◆ 关注每一次重复/迭代的执行历程

- 在递归调用之前执行的动作 **顺序输出**

- 递归调用之后的动作 **逆序输出**

注：这里的“序”指**调用**顺序



北京大学



## 方法总结之三

### ■ 如何利用递归解决问题？

◆ 如何找到 第  $i$  次做与第  $i-1$  次做之间的关系？

- 找到 两次之间的逻辑关系

- 找到 体现两次之间不同的变化因素

- 将 变化因素 体现为 函数参数



北京大学



# 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：
  - 如表达式  $2 + 3$  的逆波兰表示法为  $+ 2 3$ 。
  - 如  $(2 + 3) * 4$  的逆波兰表示法为  $* + 2 3 4$ 。
- ◆ 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： $* + 11.0 12.0 + 24.0 35.0$

■ 输出：**1357.0**



北京大学



# 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;
}
```



北京大学



## 方法总结之三

### ■ 如何利用递归解决问题？

#### ◆ 给出对一个问题的描述/即函数

- 假设有一个函数 `notation()` 能够返回 逆波兰 表达式的值

#### ◆ 解析一个问题

写出 如何利用 已经解决的子问题 解决父问题



北京大学





# 递归问题解法小结

## ■ 面对一个问题时：

- ① 从简单情况开始分析问题，找出解决问题的规律；
- ② 总结并抽取出解决方案中**反复做的事情**；
- ③ 用一个**函数（原型）**来描述这件**反复做的事情**；
- ④ 假设反复做的事情已经由上述函数实现，写出如何利用上述函数解决整体问题的步骤；
- ⑤ 分析并写出边界条件；



北京大学





# 最大公约数问题



北京大学



# 最大公约数

## ■ 问题

- ◆ 请编写程序求解两个整数 $a, b$ 的最大公约数;
- ◆ 提示:
  - **Great Common Divisor (GCD)**
  - 求 $a, b$ 的最大公约数的函数可以命名为 $\text{gcd}(a, b)$



北京大学



# 如何求最大公约数

## ■ 辗转相除法

- ◆ 又名欧几里德算法（**Euclidean algorithm**）
- ◆ 它是已知最古老的算法, 其可追溯至前**300**年。  
它首次出现于欧几里德的《几何原本》（第**VII**卷，命题**i**和**ii**）
- ◆ 在中国可以追溯至东汉出现的《九章算术》。  
—— 两个整数的最大公约数等于其中较小的数和两数的差的最大公约数



北京大学



# 辗转相除法

## ■ 原理:

◆ 若  $a > b$  的余数, 则:

$a, b$  的最大公约数 =  $b, a-b$  的最大公约数

如:  $27, 15 \Rightarrow 15, 12$

$15, 12 \Rightarrow 12, 3$

$12, 3 \Rightarrow 3, 9$

◆ 若 两数之差  $>$  较小的数, 算法结束;

◆ 较小的数 即为答案



北京大学



# 辗转相除法——另一种表述

## ■ 原理:

◆ 若  $r$  是  $a \div b$  的余数, 则  $\gcd(a, b) = \gcd(b, r)$

如:  $\gcd(27, 15) = \gcd(15, 12)$

$\gcd(15, 12) = \gcd(12, 3)$

这时,  $12 \div 3$  余数为 0, 则 答案为 3

◆ 若  $r = 0$ , 算法结束;  $b$  即为答案



北京大学



# 辗转相除法

- 用递归实现的代码:

```
int gcd(int a, int b) //求a, b的最大公约数
{
    if(a%b == 0)
        return b;
    else
        return gcd(b, a%b);
}
```



北京大学



## 如何求最小公倍数？

- 两个数的乘积等于这两个数的最大公约数与最小公倍数的积
  - ◆ Great Common Divisor (GCD)
  - ◆ Least Common Multiple (LCM)
- 即
  - ◆ 设两个数 $a, b$ 的最小公倍数为 $\text{lcm}(a, b)$
  - ◆ 设两个数 $a, b$ 的最大公约数为 $\text{gcd}(a, b)$
  - ◆ 则： $a * b = \text{gcd}(a, b) * \text{lcm}(a, b)$
- 因此，可先求 $\text{gcd}(a, b)$  再求  $\text{lcm}(a, b)$



北京大学





# 汉诺塔问题

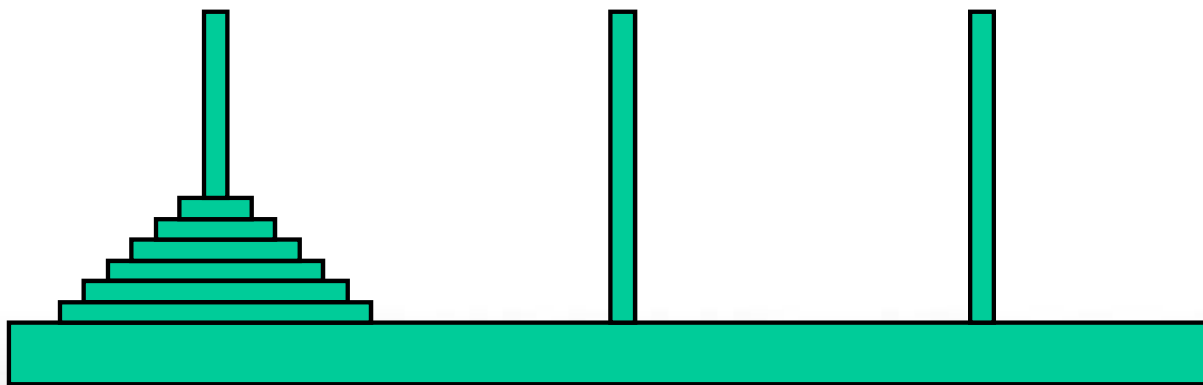


北京大学



# 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，原来他是想把**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。



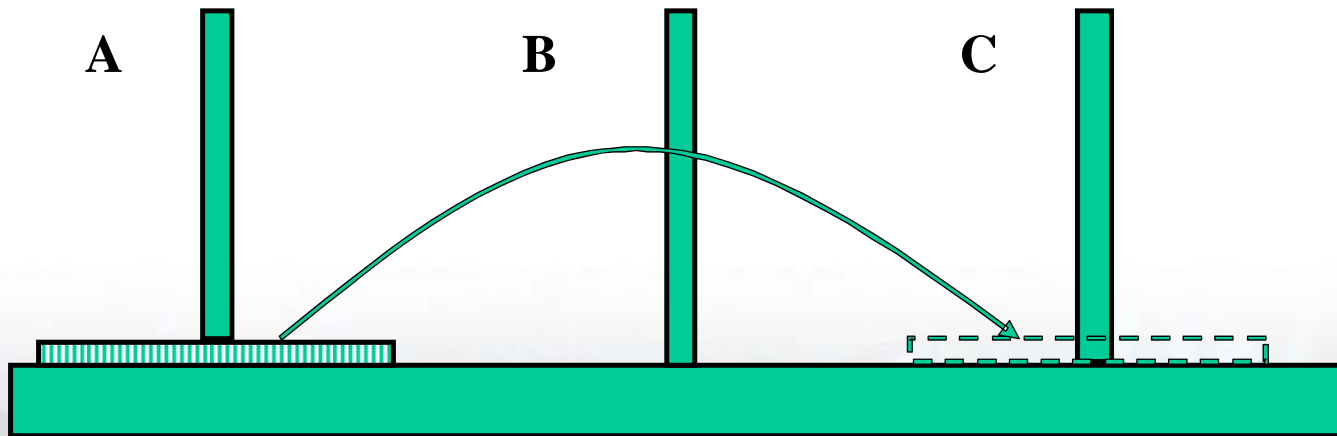
北京大学



# 汉诺塔问题

怎样编写这种程序？从思路上还是先从最简单的情况分析起，搬一搬看，慢慢理出思路。

- 1、在A柱上只有一只盘子，假定盘号为1，这时只需将该盘从A搬至C，一次完成，记为move 1 from A to C



北京大学

2、在A柱上有二只盘子，1为小盘，2为大盘。

第（1）步将1号盘从A移至B；

第（2）步将2号盘从A移至C；

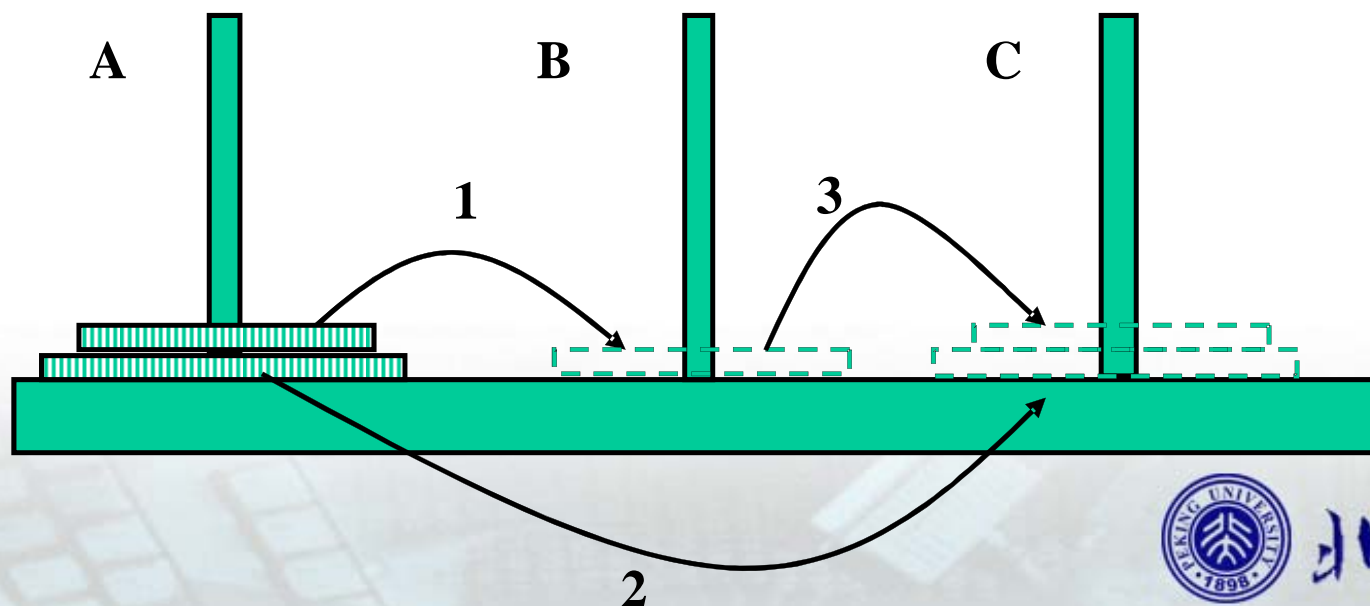
第（3）步再将1号盘从B移至C；

这三步记为：

move 1 from A to B;

move 2 from A to C;

move 3 from B to C;



北京大学

3、在A柱上有3只盘子，从小到大分别为1号，2号，3号

第（1）步将1号盘和2号盘视为一个整体；先将二者作为整体从A移至B。这一步记为

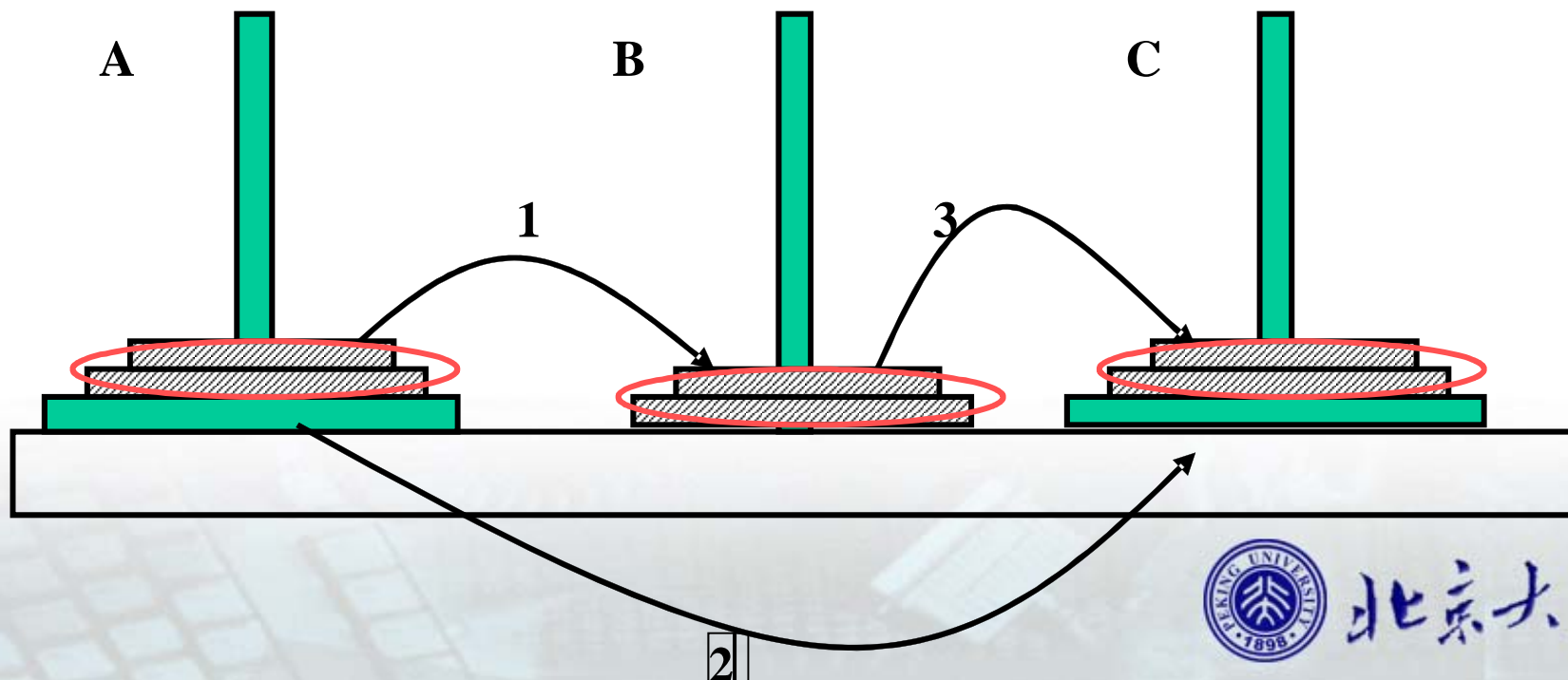
$\text{move}(2, A, C, B)$

第（2）步将3号盘从A移至C，一次到位。记为

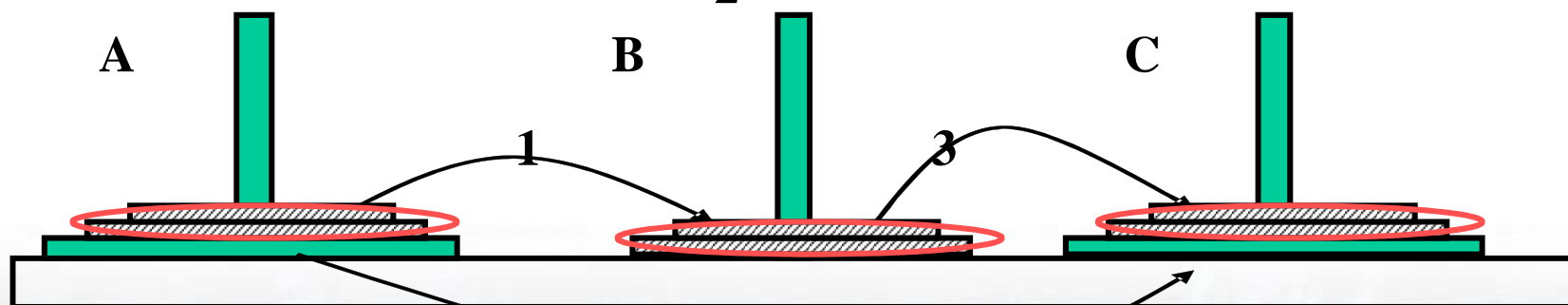
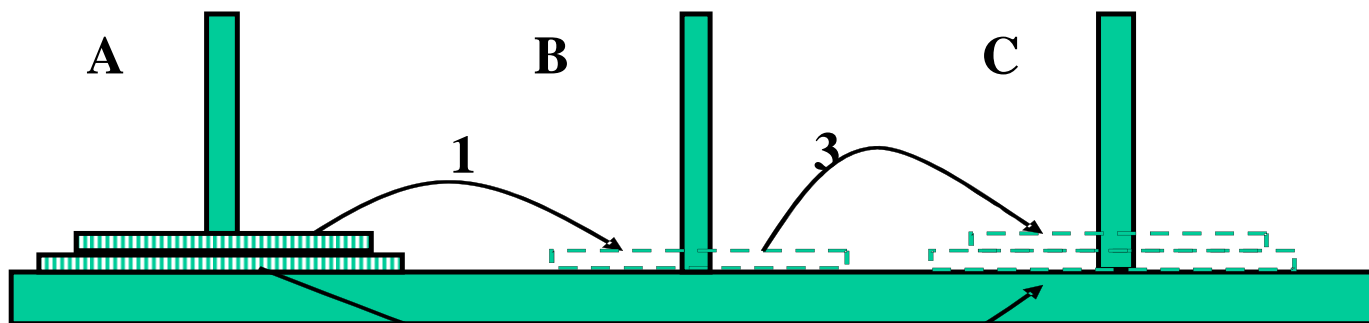
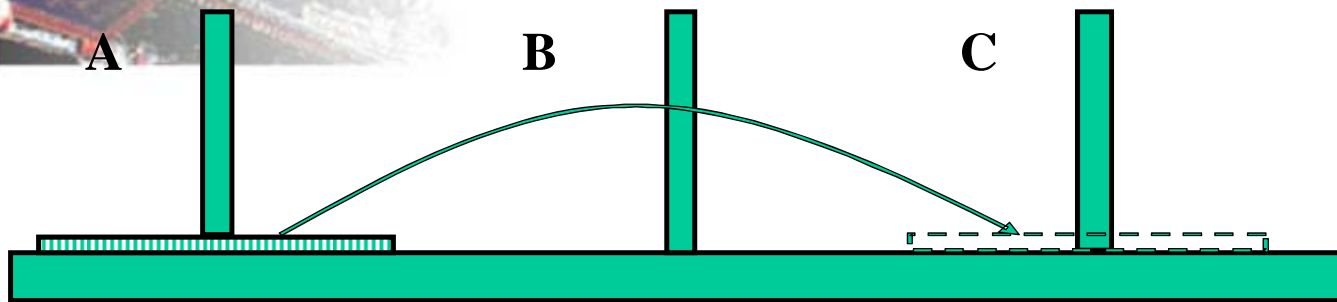
$\text{move } 3 \text{ from } A \text{ to } C$

第（3）步处于B上的作为一个整体的2只盘子，再移至C。这一步记为

$\text{move}(2, B, A, C)$



北京大学



2



北京大学



# 递归经典问题——汉诺塔问题

## ■ $\text{move}(n, A, B, C)$ 分解为3步

- ◆  $\text{move}(n-1, A, C, B)$  将上面的 $n-1$ 只盘子作为一个整体从A经C移至B;
- ◆ 输出 $n$ : A to C, 将 $n$ 号盘从A移至C, 是直接可解结点;
- ◆  $\text{move}(n-1, B, A, C)$  将上面的 $n-1$ 只盘子作为一个整体从B经A移至C。



北京大学



```

#include<iostream>
using namespace std;
void move(int m, char A, char B, char C) //表示将m个盘子从A经过B移动到C
{   if (m==1)                               //如果m为1,则为直接可解结点
    {
        cout<<"move 1# from"<<A<<" to "<<C<<endl; //直接可解结点
    }
    else                                   //如果不为1,则要调用move(m-1)
    {
        move(m-1,A,C,B);                //递归调用move(m-1)
        cout<<"move 1# from"<<A<<" to "<<C<<endl; //直接可解结点
        move(m-1,B,A,C);                //递归调用move(m-1)
    }
}
int main() {
    int n;                                //整型变量, n为盘数
    cout<<"请输入盘数n="<<endl;
    cin >> n;                             //输入盘子数目正整数n
    cout<<"在3根柱子上移"<<n<<"只盘的步骤为:"<<endl;
    move(n,'a','b','c');                  //调用函数 move(n,'a','b','c')
    return 0;
}

```



# 快速排序



北京大学



# 快速排序

## ■ 问题:

- ◆ 设计程序对整数数组中的数进行由小到大的排序;

## ■ 算法优劣的标准

- ◆ 时间代价
- ◆ 空间代价

## ■ 思考:

- ◆ 排序程序中, 时间代价、空间代价花费在哪里?

	i=1	i=2	i=3	i=4	i=5	i=6
	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
初始值	1	8	3	2	4	9
1<8; 1, 8互换	1↔8	3	2	4	9	
1<3; 1, 3互换	8	1↔3	2	4	9	
1<2; 1, 2互换	8	3	1↔2	4	9	
1<4; 1, 4互换	8	3	2	1↔4	9	
1<9; 1, 9互换	8	3	2	4	1↔9	
1到达位置	8	3	2	4	9	①
8>3; 顺序不动	8	3	2	4	9	1
3>2; 顺序不动	8	3	2	4	9	1
2<4; 2, 4互换	8	3	2↔4	9	1	
2<9; 2, 9互换	8	3	4	2↔9	1	
2到达位置	8	3	4	9	②	1



北京大学



# 快速排序

## ■ 思路:

- ◆ 假设你的手里有一把牌：要尽可能少移动牌的位置；尽可能少的进行比较；



北京大学



## 快速排序

	3	2	4	7	5	1	6
3		2	4	7	5	1	6
3	1	2	4	7	5		6
3	1	2		7	5	4	6
	1	2	3	7	5	4	6



北京大学



# 快速排序

## ■ 完成一趟排序：

- ◆ 将数组第一个元素取出，作为分界值存放到 $k$ 里。此时数组第一个元素位置空闲，用一个左探针 $L$ 指示。
- ◆ 设右探针 $R$ ，从右往左走，寻找小于 $k$ 的数。找到则将该数存放在左探针 $L$ 所指示的空闲位置，此时右探针 $R$ 所指示的位置变为空闲位置。
- ◆ 左探针 $L$ 从左往右走，寻找大于 $k$ 的数。找到则将该数放到右探针 $R$ 所指示的空闲位置，此时左探针所指示的位置变为空闲位置。
- ◆ 循环执行以上2步，直到左右探针相遇为止。左右探针相遇意味着找完所有的元素。它们相遇的地方就是 $k$ 的位置。



北京大学

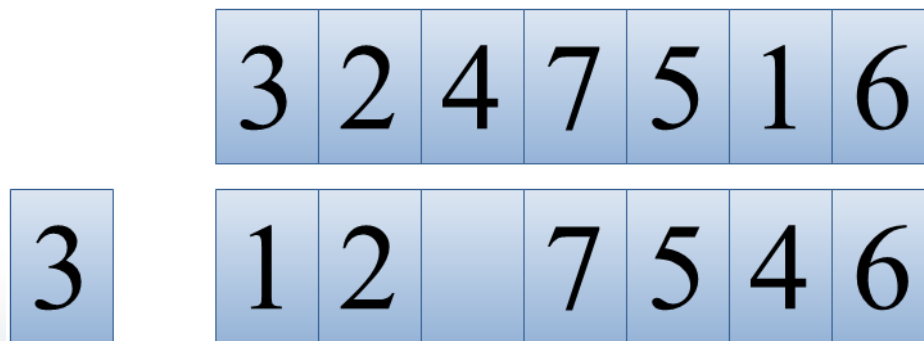




# 快速排序

## ■ 分而治之解决全部排序：

- ◆ 选定数组中的某个数字作为参照物开始排序；
- ◆ 首先以选定的数字为参照物，将要排序的数据分成两大部分，一部分的所有数据都比参照数字小，另外一部分的所有数据都比参照数字大；
- ◆ 然后再按此方法对这两部分数据分别进行排序，直到整个数据变成有序序列。



北京大学





# 快速排序

## 快速排序 (*QuickSort*)

Cooling

本例演示只说明一次划分过程  
*Partition*。红色显示的元素表示  
待排序的无序区。

R[  ]

请输入待排序的记录数组  $R[low..high]$   
(数据之间用半角逗号隔开)



Clear



Start



清华大学



# 快速排序

- 对数组是 $a[1].....a[n]$ ，一趟快速排序的算法：
  1. 设置两个变量 $L$ 、 $R$ ，排序开始的时候 $L = 1$ ， $R = n$ ;
  2. 以第一个数组元素作为关键数据，赋值给 $k$ ，即 $k = a[1]$ ;
  3. 从 $R$ 开始向前搜索，即由后开始向前搜索（ $R = R-1$ ），找到第一个小于 $k$ 的值，两者交换；
  4. 从 $L$ 开始向后搜索，即由前开始向后搜索（ $L = L+1$ ），找到第一个大于 $k$ 的值，两者交换；
  5. 重复第3、4步，直到 $L=R$ ;
  6. 将选出的 $k$ 归位， $a[L] = k$ ；（或 $a[R] = k$ ；）



```
L = LP;      R = RP;      k = array[L];  
              // array[L]给了k, L处空缺;  
do {  
    while ((L < R) && (array[R] >= k))  
        R = R - 1;    //找到右起第一个比k小的数  
    if (L < R) {  
        array[L] = array[R]; //array[R]送给array[L];  
        L ++;  
    }  
    while ((L < R) && (array[L] <= k))  
        L = L + 1;    //找到左起第一个比k大的数  
    if (L < R) {  
        array[R]=array[L];  
        R --;  
    }  
} while (L != R);  
array[L] = k;    //将最初选出的数字归位
```

```
void sort(int array[ ], int LP, int RP)
```

```
{  
    int L, R, i, k;  
    if (LP < RP) {  
        L = LP; R = RP; k = array[L];           // array[L]给了k, L处空缺;  
        do {           while ((L < R) && (array[R] >= k))  
            R = R - 1;  
            if (L < R) {  
                array[L] = array[R];           //array[R]送给array[L];  
                L ++;  
            }  
            while ((L < R) && (array[L] <= k))  
                L = L + 1;           //左边的元素<=k,让L往中间移;  
            if (L < R) {  
                array[R]=array[L];  
                R --;  
            }  
        } while (L != R);  
        array[L] = k;  
        for(i = LP; i <= RP; i = i + 1)  
            cout<<"a["<<i<<"]= "<<array[i];  
            cout << endl;  
        sort(array, LP, L-1);  
        sort(array, L + 1, RP);  
    }  
}
```



# 快速排序

```
#include <iostream>
int main()
{
    int a[10], i;
    cout << "请输入10个整数\n";
    for (i=0; i<10; i=i+1)
        cin >> a[i];
    sort(a, 0, 9); //调用sort函数, 实际参数为数组a和0, 9
    cout << "排序结果为:";
    for (i=0; i<10; i=i+1)
        cout << a[i];
    cout << endl;
    return 0; }
```



北京大学



好好想想,有没有问题?

谢谢!



北京大学