

程序设计基础

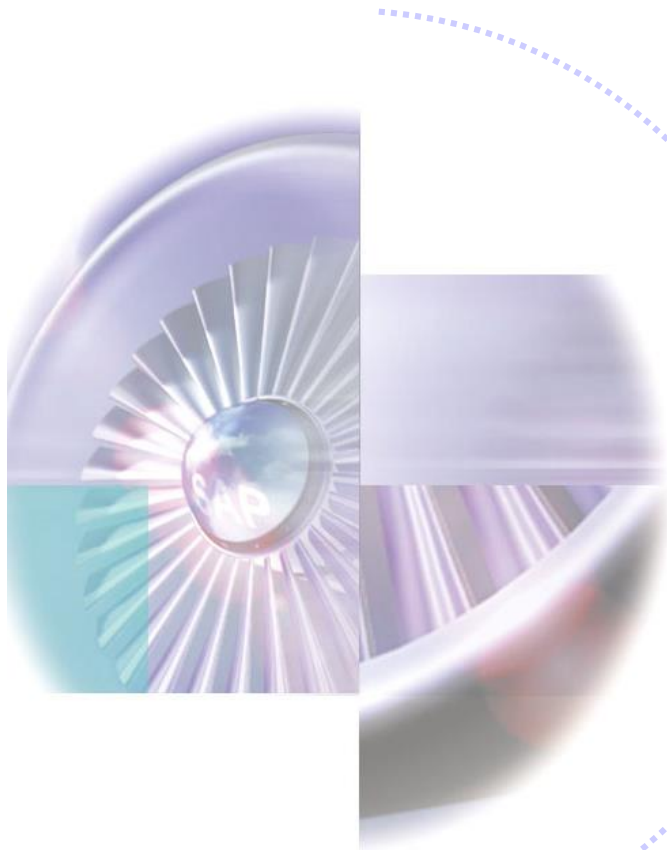
Fundamental of Programming

清华大学软件学院

刘玉身

liuyushen@tsinghua.edu.cn

Lecture 8: 递归算法

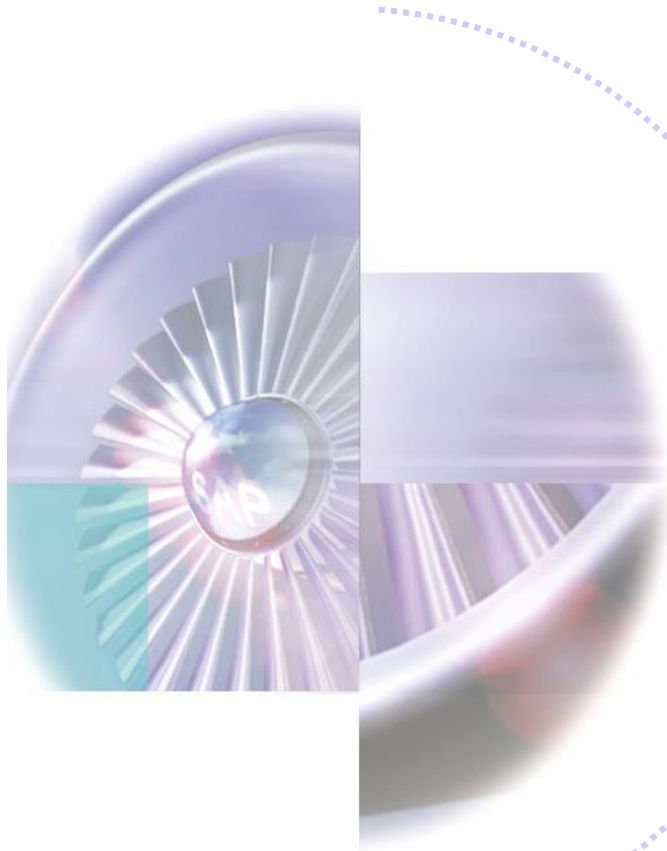


① 基本概念

② 基于分治策略的递归

③ 基于回溯策略的递归

Lecture 8: 递归算法



1 基本概念

2 基于分治策略的递归

3 基于回溯策略的递归

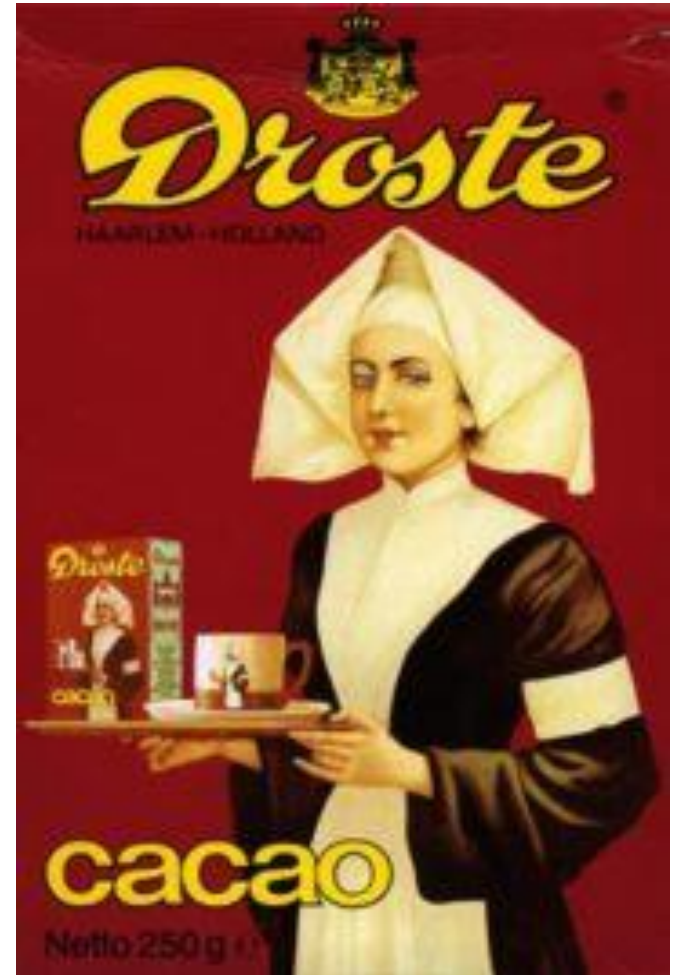
从前有座山，山上有座庙，庙里有一个老和尚
和一个小和尚，老和尚正在给小和尚讲故事。
讲的是什么呢？他说，从前.....



✦ Recursion

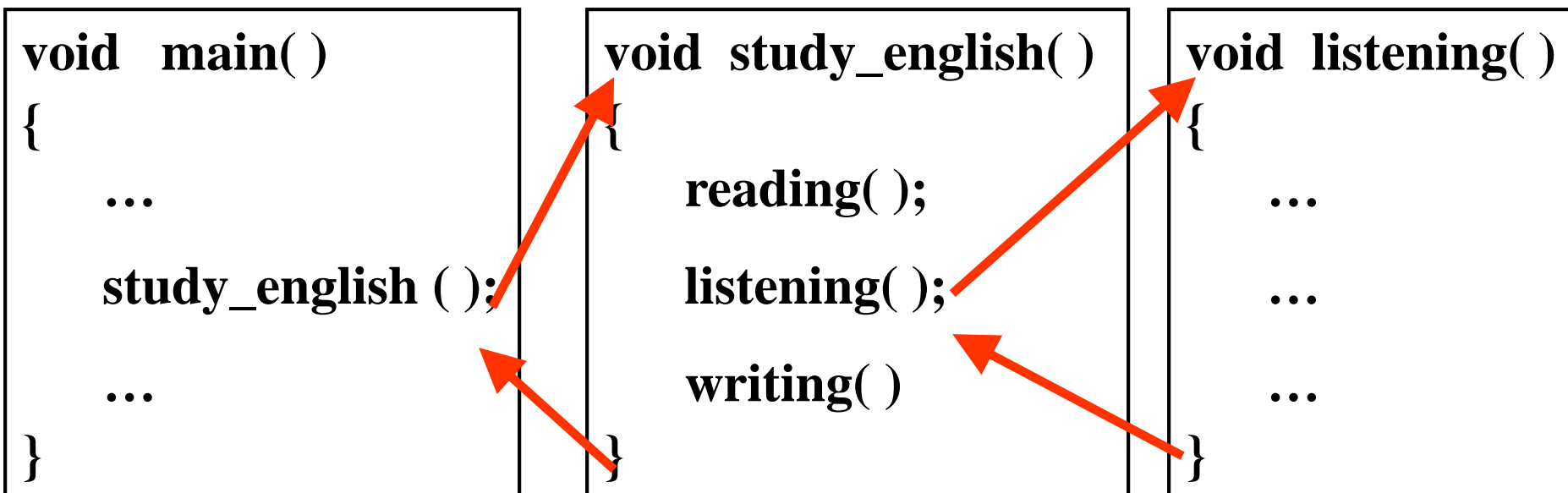
- See "*Recursion*".

✦ "*In order to understand recursion, one must first understand recursion.*"



函数的嵌套调用


C语言允许嵌套地调用函数，也就是说，在调用一个函数的过程中，又去调用另一个函数。



函数的递归调用

函数的嵌套调用有一个特例，即**递归调用**，也就是说，在调用一个函数的过程中，又出现了直接或间接地去**调用该函数本身**。

```
void tell_story()  
{  
    int old_monk, young_monk;  
    tell_story (); // tell_story 函数的递归调用  
}
```



```
void tell_story( )
{
    static int old_monk, young_monk;

    old_monk = old_monk + 1; // 年龄大了一岁
    young_monk = young_monk + 1;
    if(old_monk <= 60)           // 递归形式
        tell_story ( );
    else
        printf("对不起，已退休！"); // 递归边界
}
```


如何编写递归程序？

◆ 在语法上（简单）

☺ 递归即为普通的函数调用。

◆ 在算法上（难）

☺ 如何找到递归形式？

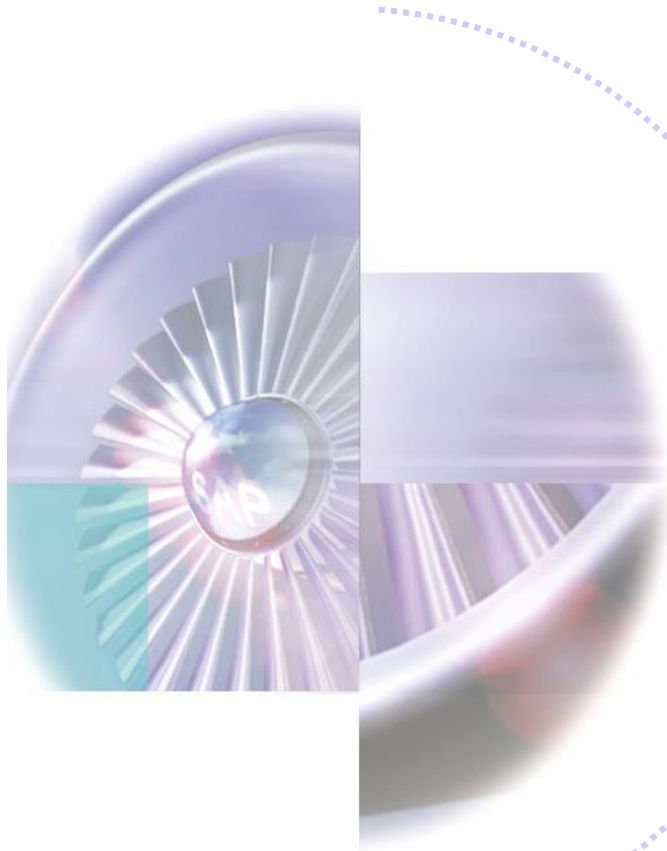
☺ 如何找到递归边界？

递归算法的类型

递归算法可以分为两种类型：

- 基于分治策略的递归算法；
- 基于回溯策略的递归算法。

Lecture 8: 递归算法



1 基本概念

2 基于分治策略的递归

3 基于回溯策略的递归

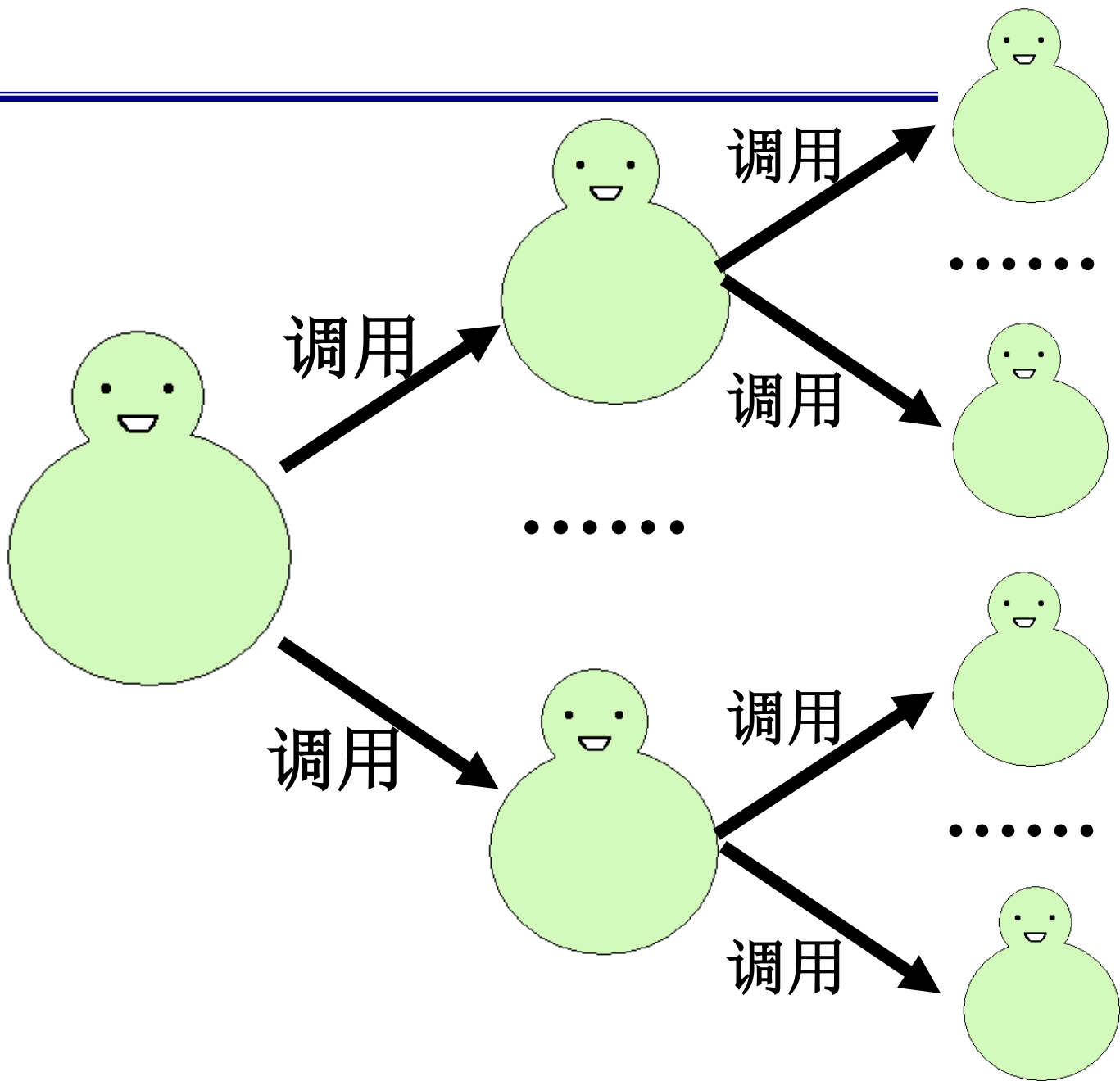
8.2.1 分而治之

分而治之 (divide-and-conquer, D&C) 的算法设计思想:

1. **Divide:** 把问题划分为若干个子问题;
2. **Conquer:** 以同样的方式分别去处理各个子问题;
3. **Combine:** 把各个子问题的处理结果综合起来, 形成最终的处理结果。

俄罗斯套娃 玛特露什卡 matryoshka





俄罗斯套娃“玛特露什卡”是如何制做的？



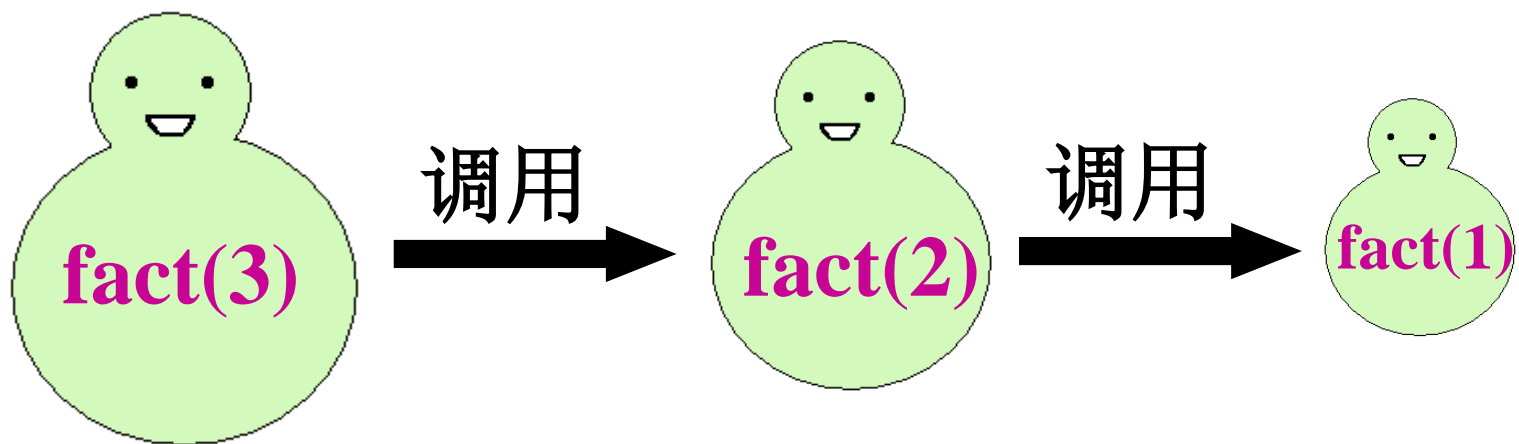
如何建立分治递归的思维方式？

☺ 基本原则：目标驱动！

☺ 计算 $n!$ ： $n! = n * (n-1)!$ ，且 $1! = 1$ 。


递归形式


递归边界



```
int main( )
{
    int n;

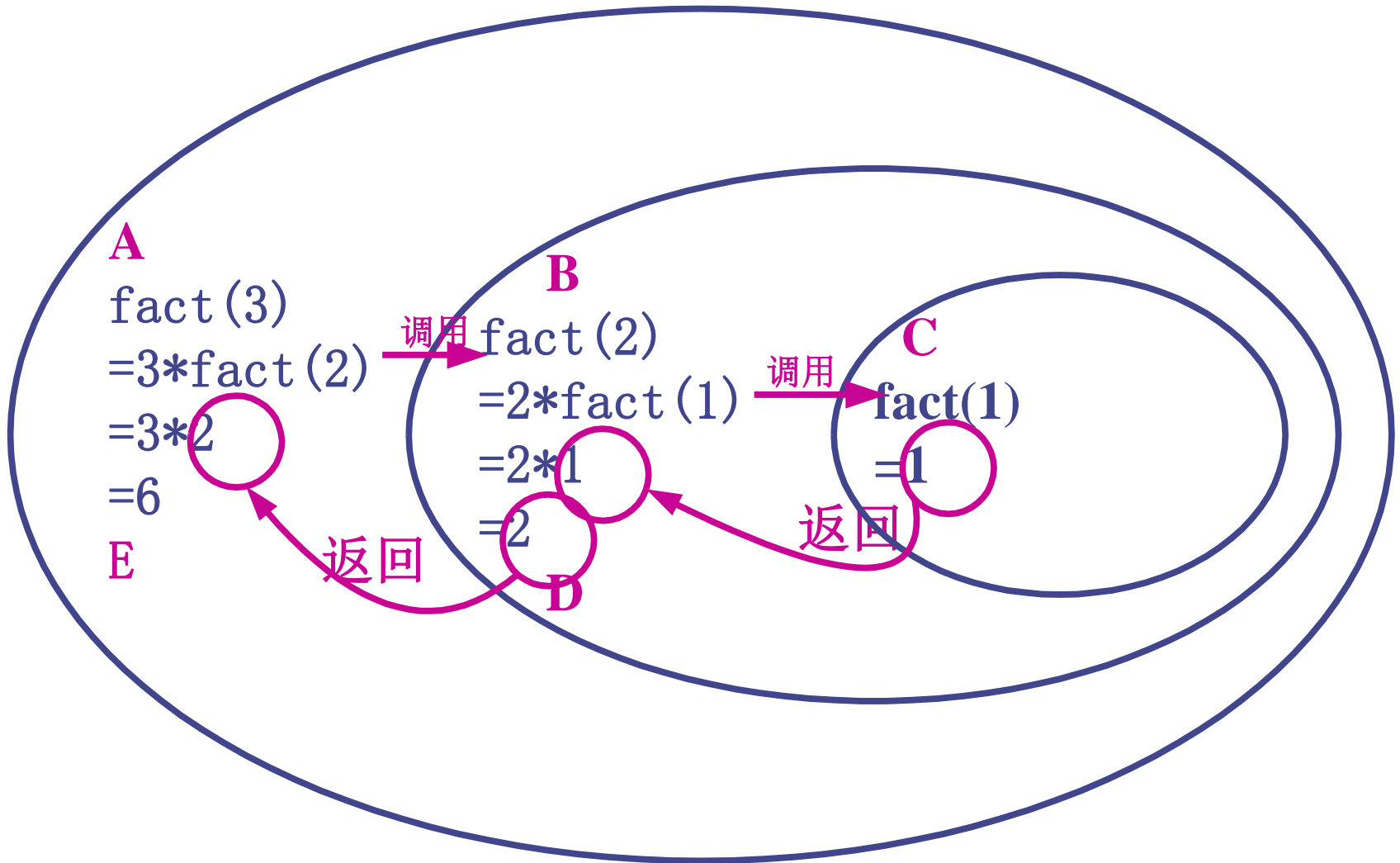
    printf("请输入一个整数: ");
    scanf("%d", &n);
    printf("%d! = %d \n", n, fact(n));
}

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

请输入一个整数: 3

3! = 6

调用和返回的递归示意图



8.2.2 寻找最大值

问题描述：

给定一个整型数组 a ，找出其中的最大值。

如何设计相应的递归算法？

如何来设计相应的递归算法？

目标： $\max\{a[0], a[1], \dots a[n-1]\}$

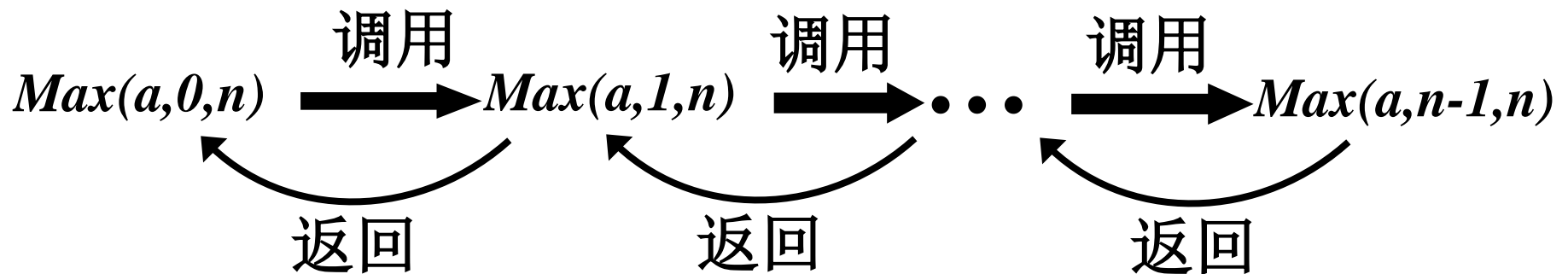
可分解为： $\max\{a[0], \max\{a[1], \dots a[n-1]\}\}$

另外已知 $\max\{x\} = x$

这就是递归算法的递归形式和递归边界，据此可以编写出相应的递归函数。

```
int Max(int a[], int first, int n)
{
    int max;

    if(first == n-1)    return a[first];
    max = Max(a, first+1, n);
    if(max < a[first])
        return a[first];
    else    return max;
}
```



8.2.3 折半查找法

问题描述：

查找（Searching）： 根据给定的某个值，在一组数据（尤其是一个数组）当中，确定有没有出现相同取值的数据元素。

顺序查找、折半查找 (binary search)。

-
- 前提：数据是有序排列的；
 - 基本思路：
 - 将目标值与数组的中间元素进行比较；
 - 若相等，查找成功。否则根据比较的结果将查找范围缩小一半，然后重复此过程。

请问：

4565926是否在此列表当中？

4120243
4276013
4328968
4397700
4462718
4466240
4475579
4478964
4480332
4494763
4499043
4508710
4549243
4563697
4564531
4565926
4566088
4572874

请问：

4565926是否在此列表当中？

4120243
4276013
4328968
4397700
4462718
4466240
4475579
4478964
4480332
4494763
4499043
4508710
4549243
4563697
4564531
4565926
4566088
4572874

← 数组正中间的元素。

请问：

4565926是否在

此列表当中？

4120243
4276013
4328968
4397700
4462718
4466240
4475579
4478964
4480332
4494763
4499043
4508710
4549243
4563697
4564531
4565926
4566088
4572874

不予考虑

请问：

4565926是否在此列表当中？

4120243
4276013
4328968
4397700
4462718
4466240
4475579
4478964
4480332
4494763
4499043
4508710
4549243
4563697
4564531
4565926
4566088
4572874

← 正中间的元素

请问：

4565926是否在此列表当中？

4120243
4276013
4328968
4397700
4462718
4466240
4475579
4478964
4480332
4494763
4499043
4508710
4549243
4563697
4564531
4565926
4566088
4572874

↑
不予考虑
↓

← 正中间的元素

```
int main()
{
    int b[] = {5, 13, 19, 21, 37, 56, 64, 75,
               80, 88, 92};
    int x = 21;

    printf("x位于数组的第%d个元素\n",
           bsearch(b, x, 0, 10));
}
```

问题分析

1. 函数原型:

int bsearch(int b[], int x, int L, int R);

查找x，成功则返回数组下标，失败返回-1

2. 递归的形式？

3. 递归的边界？

$n = 11, x = 21$

b	0	1	2	3	4	5	6	7	8	9	10
	5	13	19	21	37	56	64	75	80	88	92
	↑ L					↑ mid					↑ R

↑**L** ↑**mid** ↑**R** $x < b[mid], \text{ 令 } R = mid-1$

↑**L** ↑**R** $x > b[mid], \text{ 令 } L = mid+1$

20? ↑**mid** $x == b[mid], \text{ 查找成功}$


```
int bsearch(int b[], int x, int L, int R)
{
    int mid;

    if(L > R) return(-1);
    mid = (L + R)/2;
    if(x == b[mid])
        return mid;
    else if(x < b[mid])
        return bsearch(b, x, L, mid-1);
    else
        return bsearch(b, x, mid+1, R);
}
```

x位于数组的第3个元素

```
#include <stdio.h>
int bsearch(int b[], int n, int x)
{
    int mid, L, R;
    L = 0; R = n - 1;
    while (L <= R) {
        mid = (L + R) / 2;
        if (x == b[mid])
            return mid;
        else if (x < b[mid])    R = mid - 1;
        else                    L = mid + 1;
    }
    return (-1);
}
int main()
{
    int b[] = {5, 13, 19, 21, 37, 56, 64, 75, 80, 88, 92};
    int x = 21;
    printf("x位于数组的第%d个元素\n", bsearch(b, 10, x));
}
```

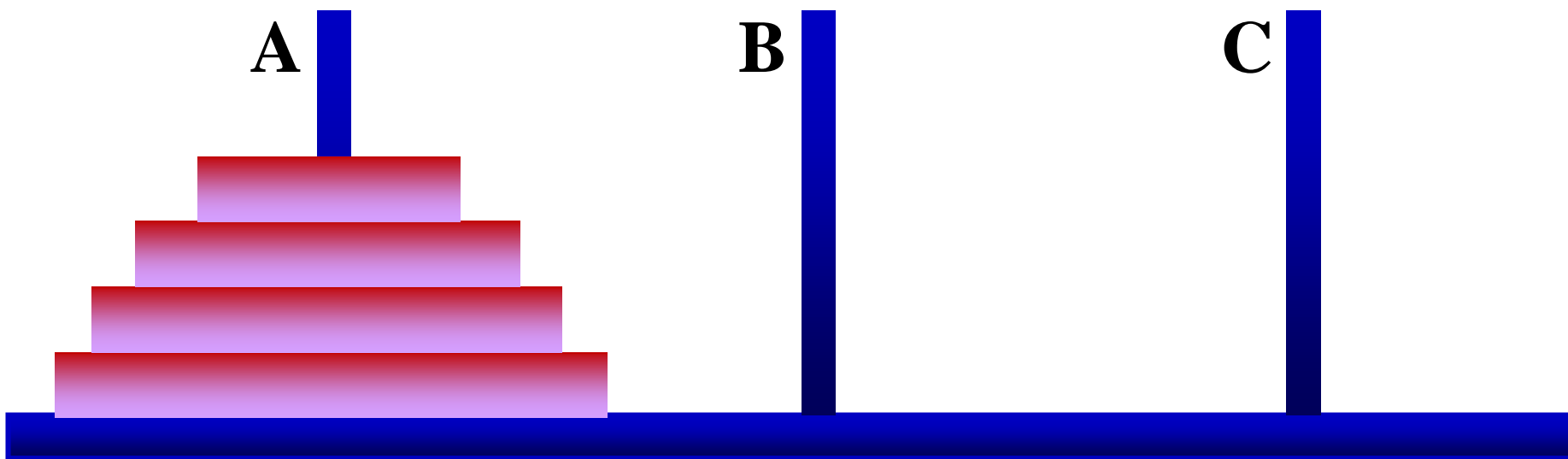
x位于数组的第3个元素

递归与循环实现对比:

1. 循环终止条件(L <= R)?
2. R和L的更新?
3. return ?

8.2.4 汉诺(Hanoi)塔问题

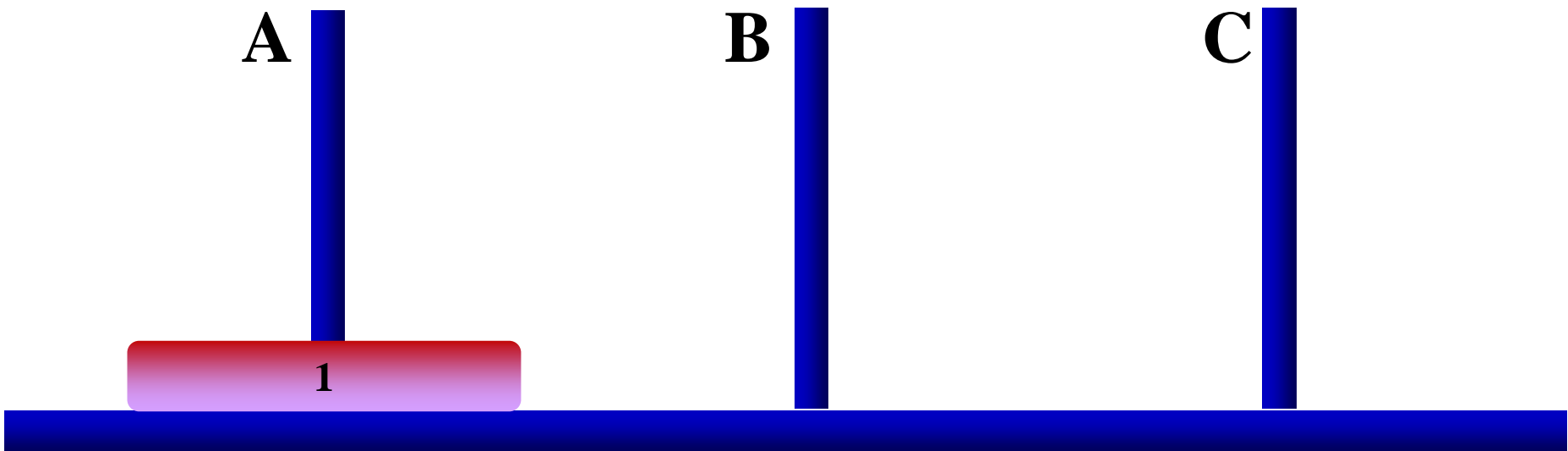
相传在古印度Bramah庙中，有位僧人整天把三根柱子上的金盘倒来倒去，原来他是想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中遵守以下规则：每次只允许移动一只盘，且大盘不得落在小盘上（简单吗？若每秒移动一只盘子，需**5800亿年**）



怎样编写这种程序？思路还是先从最简单的情况分析起，搬一搬看，慢慢理出思路。

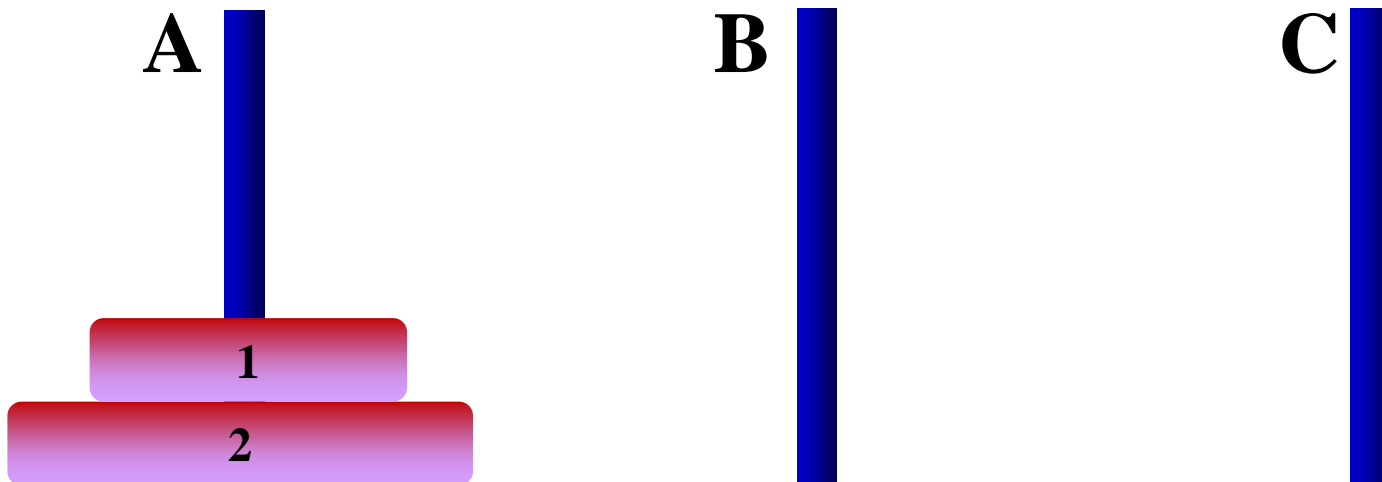
1、在A柱上只有一只盘子，假定盘号为1，这时只需将该盘直接从A搬至C，记为

move from A to C



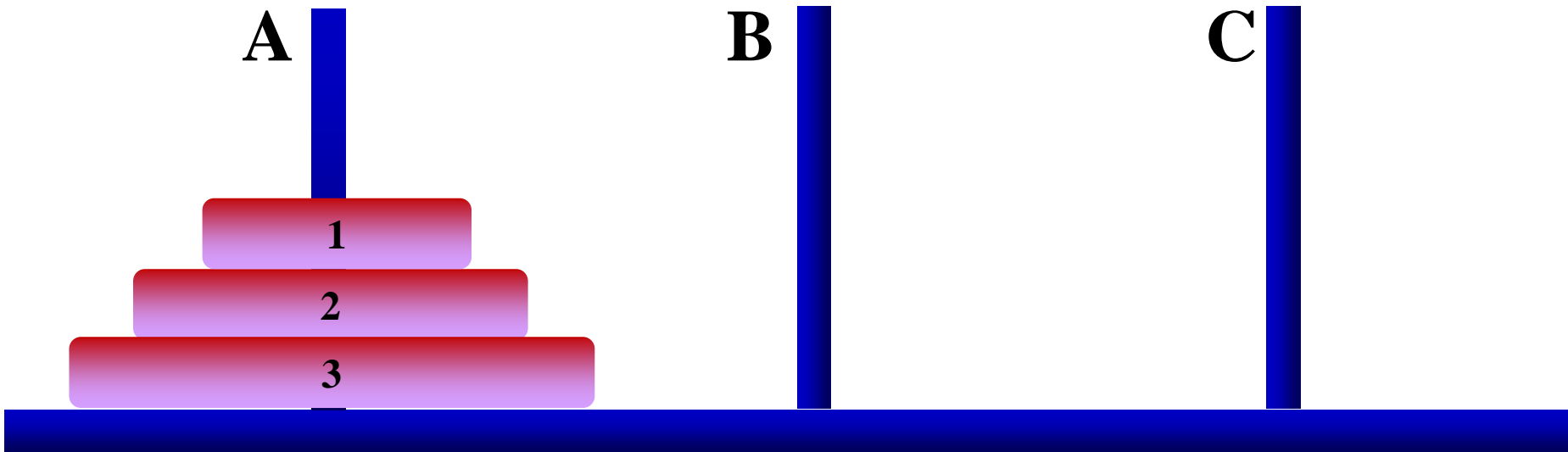
2、在A柱上有二只盘子，1为小盘2为大盘。
分三步进行：

move from A to B;
move from A to C;
move form B to C;



3、在A柱上有3只盘子，从小到大分别为1号，2号，3号。怎么移？

分七步！

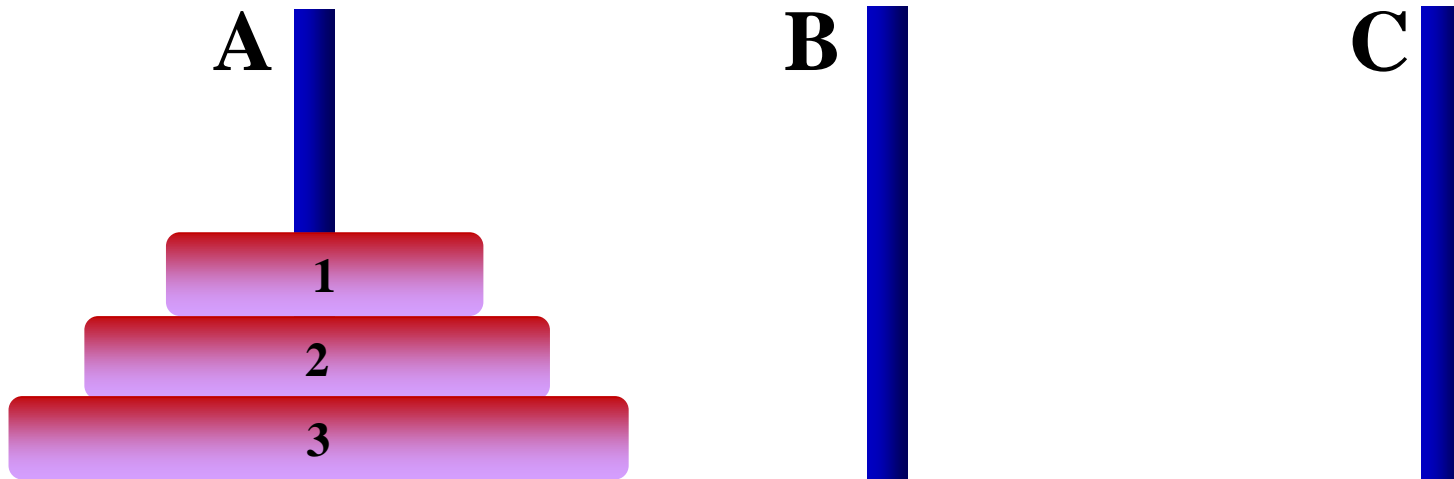


分三步进行:

move 2 discs from A to B using C;

move from A to C;

move 2 discs from B to C using A;



4、在A柱上有 n 个盘子, 从小到大分别为1号、2号、3号、...、 n 号。

第 1 步: 将1号、2号、...、 $n-1$ 号盘作为一个整体, 在C的帮助下, 从A移至B;

第 2 步: 将 n 号盘从A移至C;

第 3 步: 再将1号、2号、...、 $n-1$ 号盘作为一个整体, 在A的帮助下, 从B移至C;

这三步记为:

move $n-1$ discs from A to B using C;

move 1 discs from A to C;

move $n-1$ discs from B to C using A ;

递归形式!

定义函数 `move(int n, char A, char B, char C);`
表示

`move` **n** discs from **A** to **C** using **B**;

如果 `n = 1`，即表示

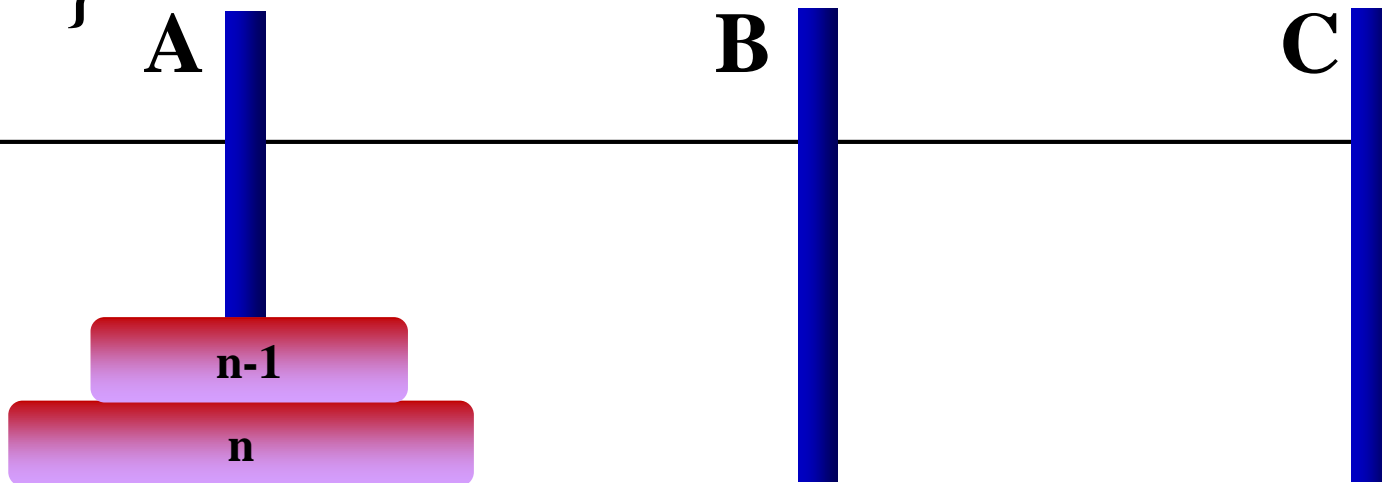
`move` from **A** to **C**; ← 移动的是谁?

```
#include <stdio.h>
void move(int n, char A, char B, char C);

int main( )
{
    int n;

    printf("请输入一个整数: ");
    scanf("%d", &n);
    move(n, 'A', 'B', 'C');
}
```

```
void move(int n, char A, char B, char C)
{
    if(n == 1)
        printf("move #1 from %c to %c\n", A, C);
    else
    {
        move(n-1, A, C, B);
        printf("move #%d from %c to %c\n", n, A, C);
        move(n-1, B, A, C);
    }
}
```



运行结果

请输入一个整数: 3

move #1 from A to C

move #2 from A to B

move #1 from C to B

move #3 from A to C

move #1 from B to A

move #2 from B to C

move #1 from A to C

请输入一个整数: 4

move #1 from A to B

move #2 from A to C

move #1 from B to C

move #3 from A to B

move #1 from C to A

move #2 from C to B

move #1 from A to B

move #4 from A to C

move #1 from B to C

move #2 from B to A

move #1 from C to A

move #3 from B to C

move #1 from A to B

move #2 from A to C

move #1 from B to C

如何统计移动的次数?

```
int main()
{
    int n, num;
    printf("请输入一个整数: ");
    scanf("%d", &n);
    num = move(n, 'A', 'B', 'C');
    printf("总计移动的次数: %d\n", num);
}
```

请输入一个整数: 5
move #1 from A to B
.....
总计移动的次数: 31

```
int move(int n, char A, char B, char C) {
    static int num = 0;
    if(n == 1) {
        printf("move #1 from %c to %c\n", A, C); num++;
    }
    else {
        move(n-1, A, C, B);
        printf("move #%d from %c to %c\n", n, A, C); num++;
        move(n-1, B, A, C);
    }
    return num;
}
```

使用static变量记录次数!

汉诺塔问题的时间复杂度

- $\text{Move}(1) = 1;$
- $\text{Move}(2) = 3;$
- $\text{Move}(3) = 7;$
-
- $\text{Move}(k+1) = 2 * \text{Move}(k) + 1; (k > 1)$
-
- $\text{Move}(n) = 2^n - 1; (n > 0)$
- if $n = 64$, then $\text{Move}(n) = 1.84467440 * 10^{19}$
- 一年 = 60秒*60分*24小时*365天 $\approx 3 * 10^7$ 秒
- 假如每秒钟移动一次, 大约需要 5800 亿年



8.2.5 青蛙过河

一条小溪尺寸不大，青蛙可以从左岸跳到右岸，在左岸有一石柱L，面积只容得下一只青蛙落脚，同样右岸也有一石柱R，面积也只容得下一只青蛙落脚。有一队青蛙从尺寸上一个比一个小。我们将青蛙从小到大，用1, 2, ..., n编号。规定**初始**时这队青蛙只能趴在**左岸的石头L**上，按编号一个落一个，小的落在大的上面。不允许大的在小的上面。在小溪中有S根**石柱**，有y片**荷叶**，规定溪中的柱子上允许一只青蛙落脚，如有多只同样要求按编号一个落一个，大的在下，小的在上，而且必须编号相邻。对于荷叶只允许一只青蛙落脚，不允许多只在其上。对于**右岸的石柱R**，与左岸的石柱L一样允许多个青蛙落脚，但须一个落一个，小的在上，大的在下，且编号相邻。当青蛙从左岸的L上跳走后就不允许再跳回来；同样，从左岸L上跳至右岸R，或从溪中荷叶或溪中石柱跳至右岸R上的青蛙也不允许再离开。问在已知溪中有S根石柱和y片荷叶的情况下，最多能跳过多少只青蛙？



这题看起来较难，但是如果我们认真分析，理出思路，就可化难为易。

思路：

1、简化问题，探索规律。先从个别再到一般，要善于对多个因素作分解，孤立出一个一个因素来分析，化难为易。

2. 定义函数

$\text{Jump} (S, y)$ —— 最多可跳过河的青蛙数

其中： S —— 河中柱子数

y —— 荷叶数

3. 先看简单情况，河中无柱子： $S = 0$ ，

$\text{Jump}(0, y)$

当 $y = 1$ 时， $\text{Jump}(0, 1) = 2$ ；

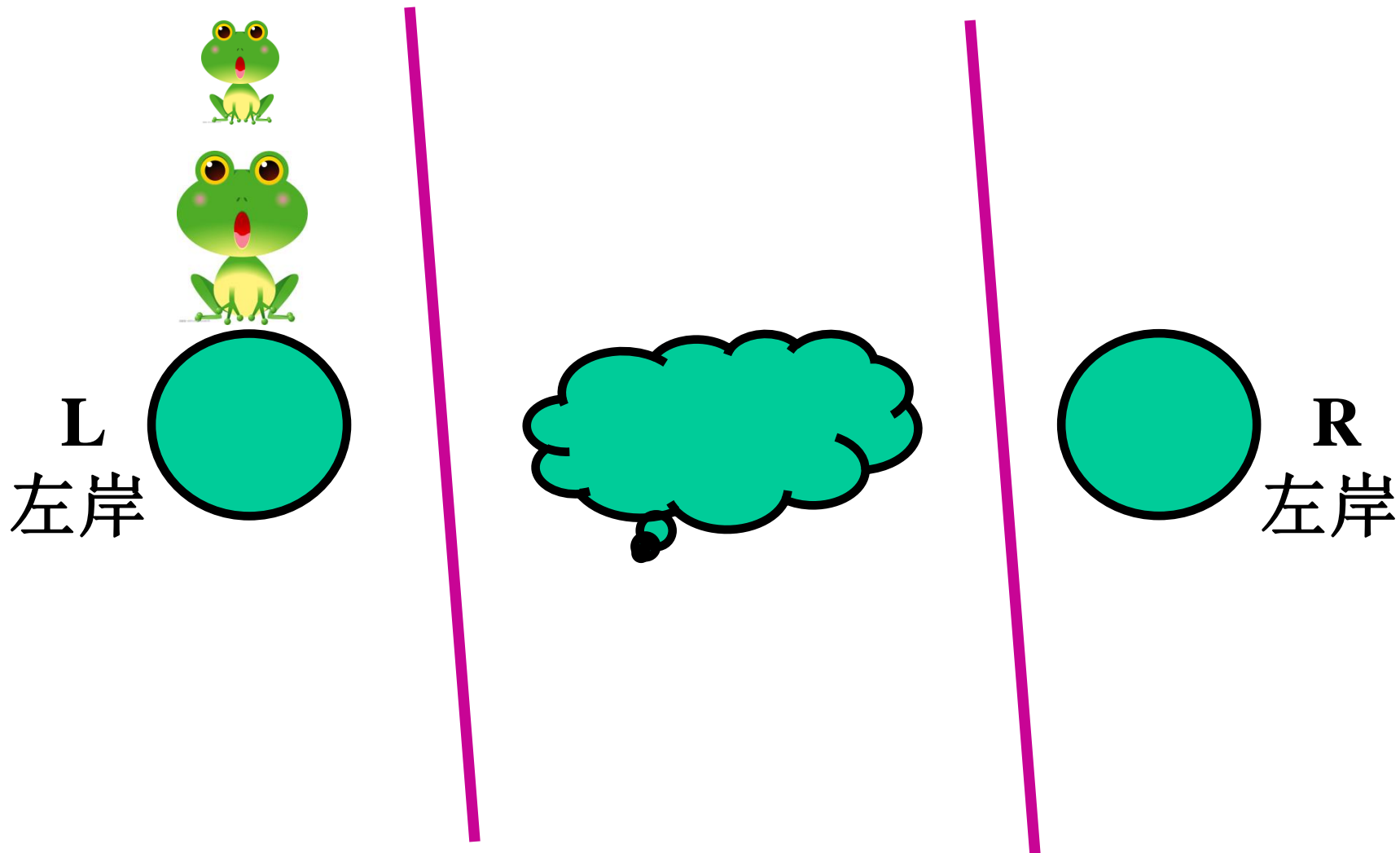
说明：河中有一片荷叶，可以过两只青蛙，起始时 L 上有两只青蛙，1# 在 2# 上面。

第一步：1# 跳到荷叶上；

第二步：2# 从 L 直接跳至 R 上；

第三步：1# 再从荷叶跳至 R 上。

如下图：



当 $y = 2$ 时,

$\text{Jump}(0, 2) = 3$;

说明: 河中有两片荷叶时, 可以过 3 只青蛙。起始时:

1#, 2#, 3# 3只青蛙落在 L 上,

第一步: 1# 从 L 跳至叶 1 上,

第二步: 2# 从 L 跳至叶 2 上,

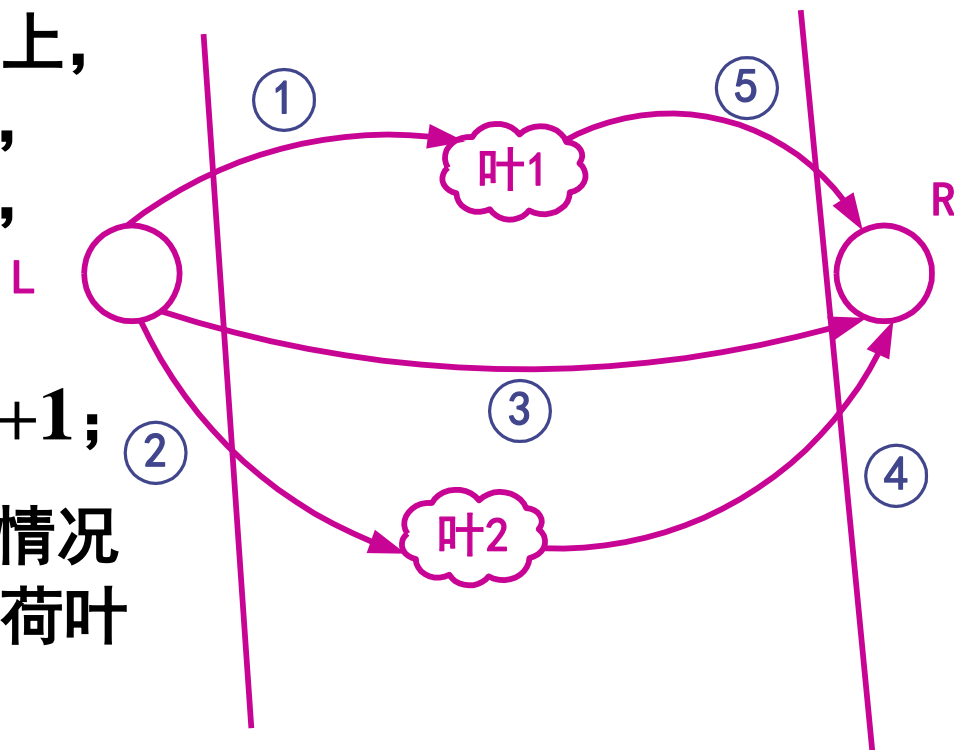
第三步: 3# 从 L 直接跳至 R 上,

第四步: 2# 从叶 2 跳至 R 上,

第五步: 1# 从叶 1 跳至 R 上,

采用归纳法: $\text{Jump}(0, y) = y + 1$;

意思是: 在河中没有石柱的情况下, 过河的青蛙数仅取决于荷叶数, 数目是荷叶数+1。

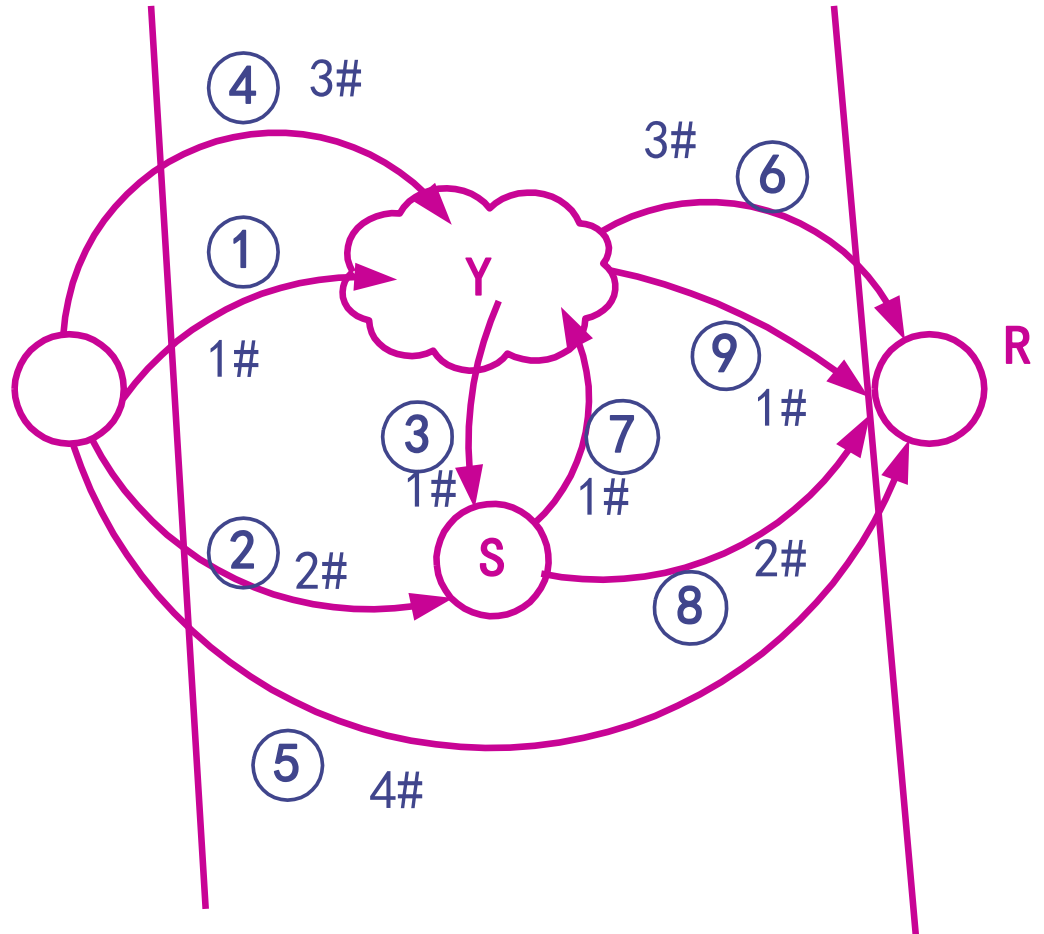


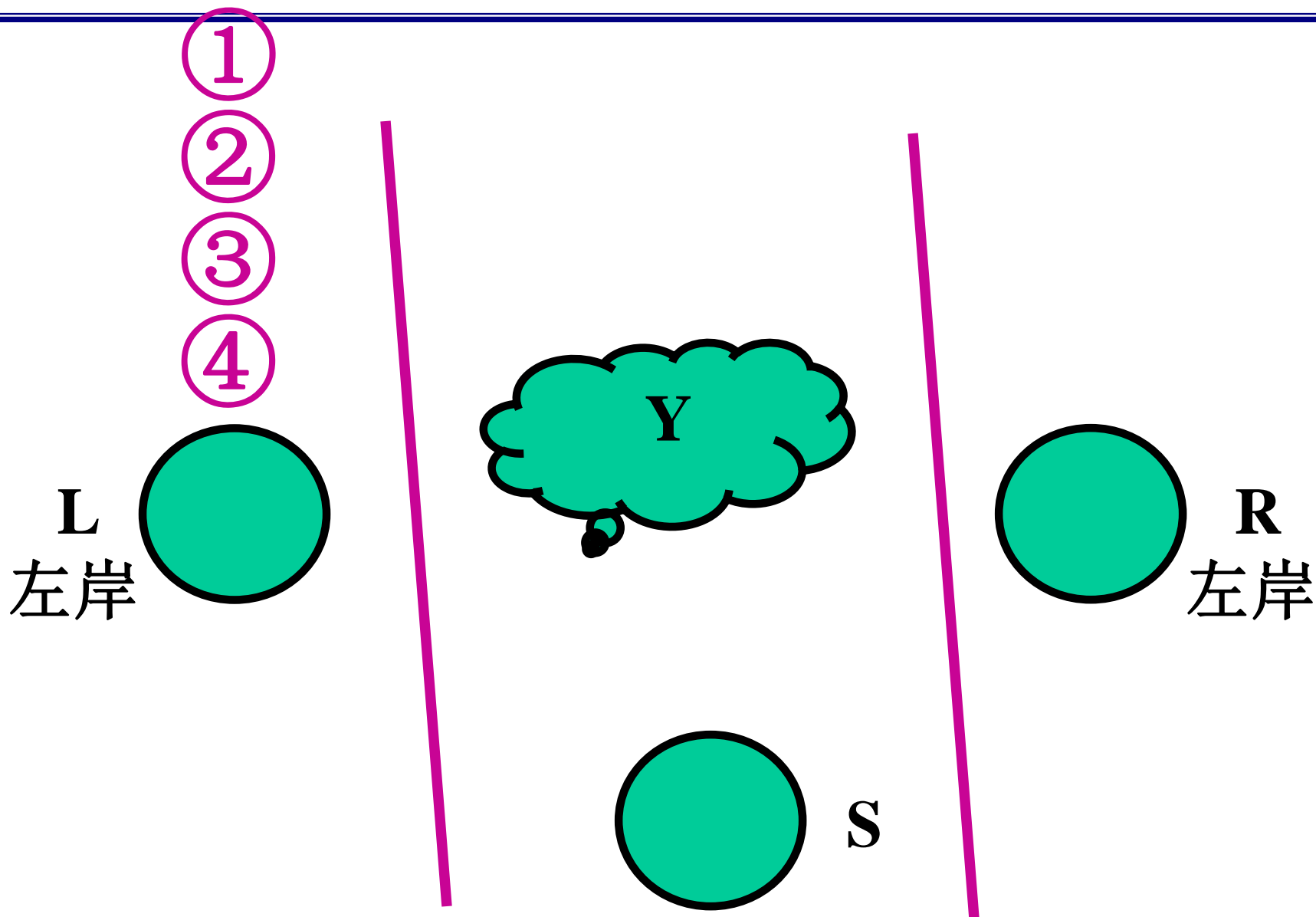
再看Jump(S, y)

先看一个最简单情况： S = 1, y = 1。

从图上看需要 9 步，跳过 4 只青蛙。

1# 青蛙从 L \rightarrow Y;
2# 青蛙从 L \rightarrow S;
1# 青蛙从 Y \rightarrow S;
3# 青蛙从 L \rightarrow Y; L
4# 青蛙从 L \rightarrow R;
3# 青蛙从 Y \rightarrow R;
1# 青蛙从 S \rightarrow Y;
2# 青蛙从 S \rightarrow R;
1# 青蛙从 Y \rightarrow R;





对于 $S = 1$, $y = 1$ 的情形, 从另外一个角度来分析

若没有石柱 S , 最多可过 $y+1 = 2$ 只青蛙。

有了石柱 S 后, 最多可过 $2*2 = 4$ 只青蛙。

步骤1:

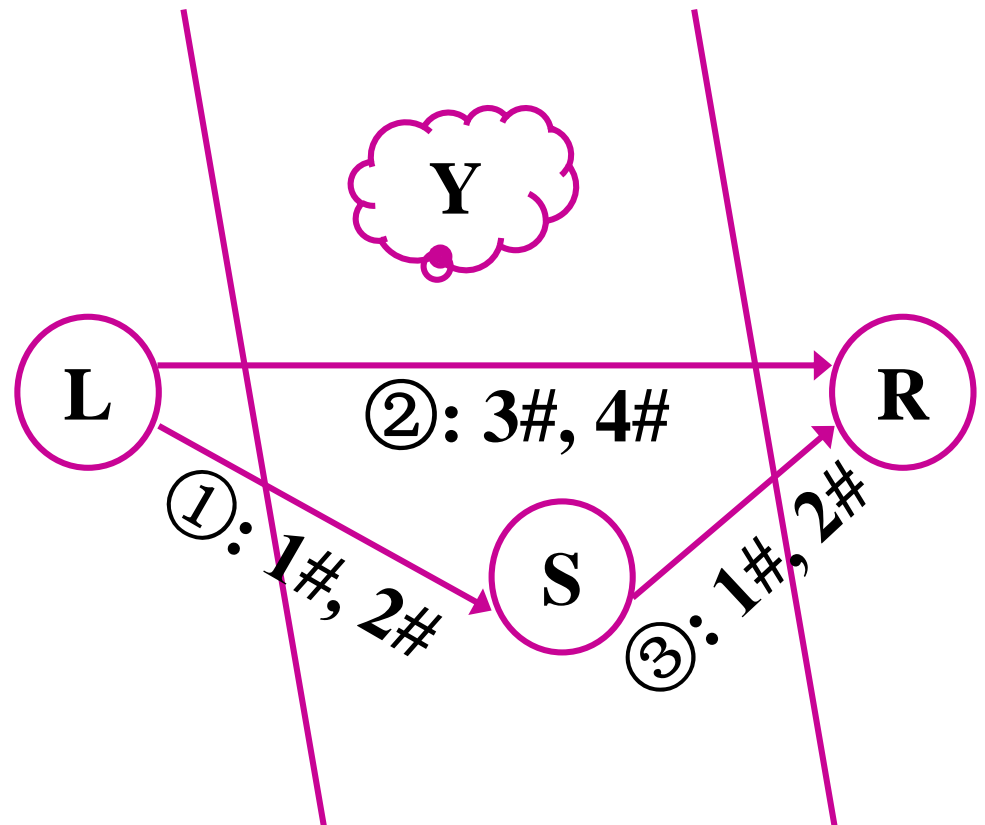
1#和2# 从 $L \xrightarrow{Y} S$;

步骤2:

3#和4# 从 $L \xrightarrow{Y} R$;

步骤3:

1#和2# 从 $S \xrightarrow{Y} R$;



对于 $S = 1$, $y > 1$ 的情形

若没有石柱 S , 最多可过 $y+1$ 只青蛙。

有了石柱 S 后, 最多可过 $2*(y+1)$ 只青蛙。

步骤1:

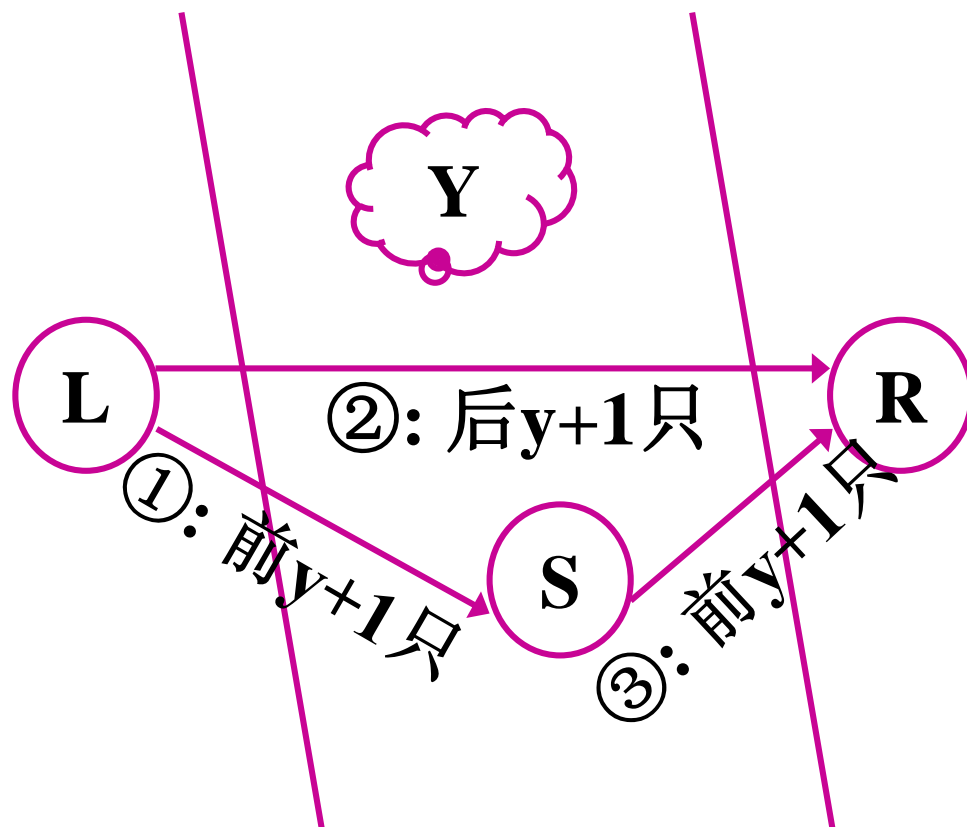
前 $y+1$ 只 从 $L \xrightarrow{Y} S$;

步骤2:

后 $y+1$ 只 从 $L \xrightarrow{Y} R$;

步骤3:

前 $y+1$ 只 从 $S \xrightarrow{Y} R$;



对于 $S = 2$, $y > 1$ 的情形

若只有石柱 $S1$, 最多可过 $2*(y+1)$ 只青蛙。

有了石柱 $S2$ 后, 最多可过 $4*(y+1)$ 只青蛙。

步骤1:

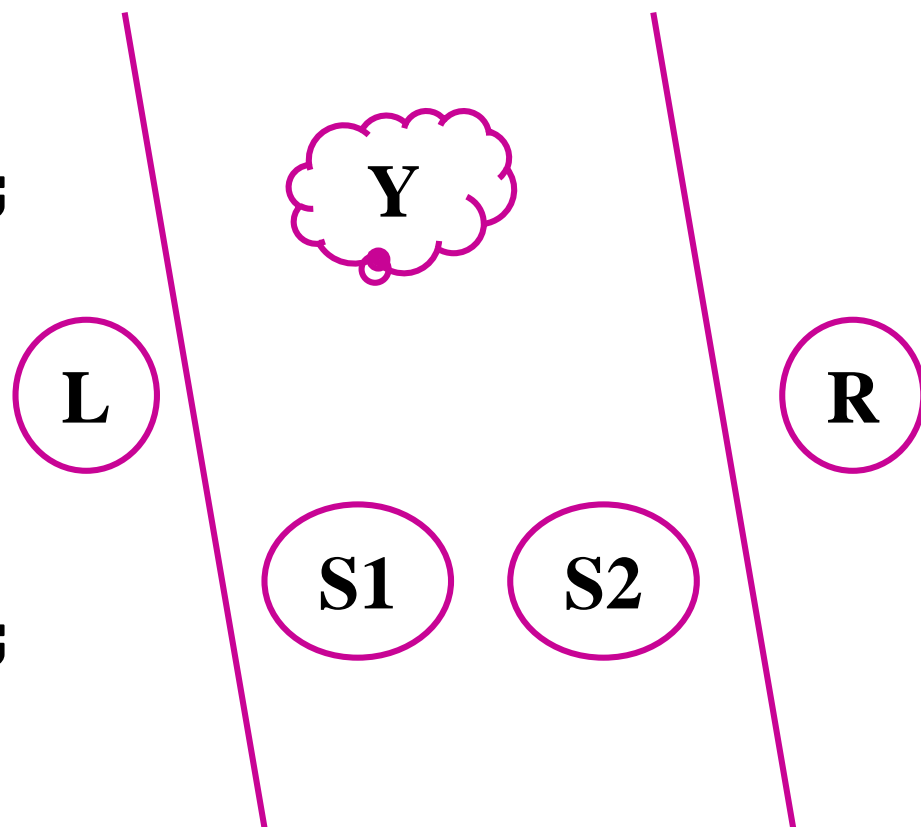
前 $2*(y+1)$ 只 从 $L \xrightarrow{Y, S1} S2$;

步骤2:

后 $2*(y+1)$ 只 从 $L \xrightarrow{Y, S1} R$;

步骤3:

前 $2*(y+1)$ 只 从 $S2 \xrightarrow{Y, S1} R$;



采用归纳法，可得出如下的递归形式：

$$\text{Jump}(S, y) = 2 * \text{Jump}(S-1, y);$$

意思是：先让一半的青蛙利用 y 片荷叶和 $S-1$ 根石柱，落在河中剩下的那根石柱上；然后让另一半的青蛙利用 y 片荷叶和 $S-1$ 根石柱，落在右岸的 R 上面；最后再让先前的一半青蛙，利用 y 片荷叶和 $S-1$ 根石柱，落在右岸的 R 上面。

递归边界： $\text{Jump}(0, y) = y + 1$

```
int main( )
{
    int S, y;

    printf("请输入石柱和荷叶的数目: ");
    scanf("%d %d", &S, &y);
    printf("最多有 %d 只青蛙过河\n", Jump(S, y));
}

int Jump(int S, int y)
{
    if(S == 0) return( y + 1 );
    return( 2 * Jump(S-1, y));
}
```

8.2.6 快速排序

快速排序的基本思路：

- 1、在数组a中，有一段待排序的数据，下标从1到r。
- 2、取a[1]放在变量value中，通过由右、左两边对value的取值的比较，为value选择应该排定的位置。这时要将比value大的数放右边，比它小的数放左边。当value到达最终位置后（如下标m），由它划分了左右两个集合[1..m-1]、[m+1..r]。然后转第1步，再用相同的思路分别去处理左集合和右集合。

令 $\text{qsort}(l, r)$ 表示将数组元素从下标为 l 到下标为 r 的共 $r-l+1$ 个元素进行从小到大的快速排序。

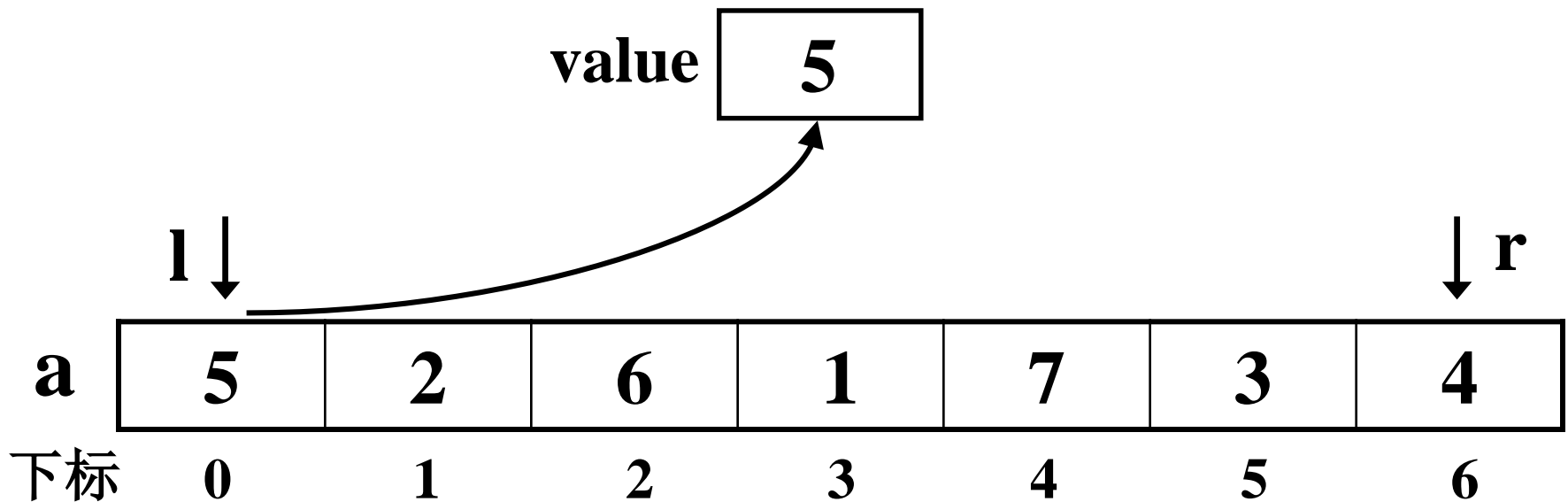
目标:

l : Left
 r : Right

- 1、让 $\text{value} = a[l]$
- 2、将 value 放在 $a[m]$ 中, $l \leq m \leq r$
- 3、使 $a[l], a[l+1], \dots, a[m-1] \leq a[m]$
- 4、使 $a[m] < a[m+1], a[m+2], \dots, a[r]$

例子：数组a当中有7个元素等待排序。

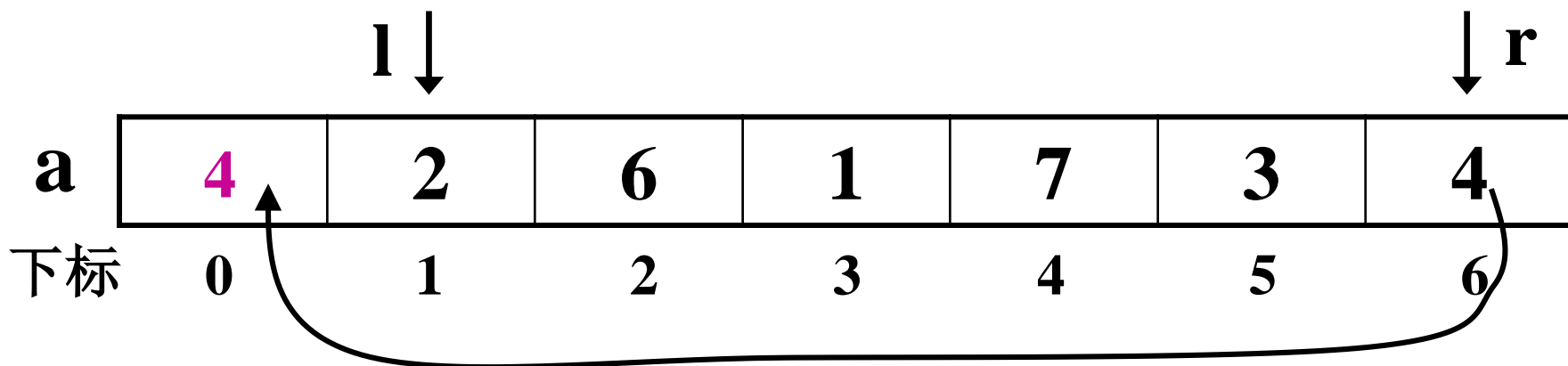
首先，让 $\text{value} = a[l] = a[0] = 5$



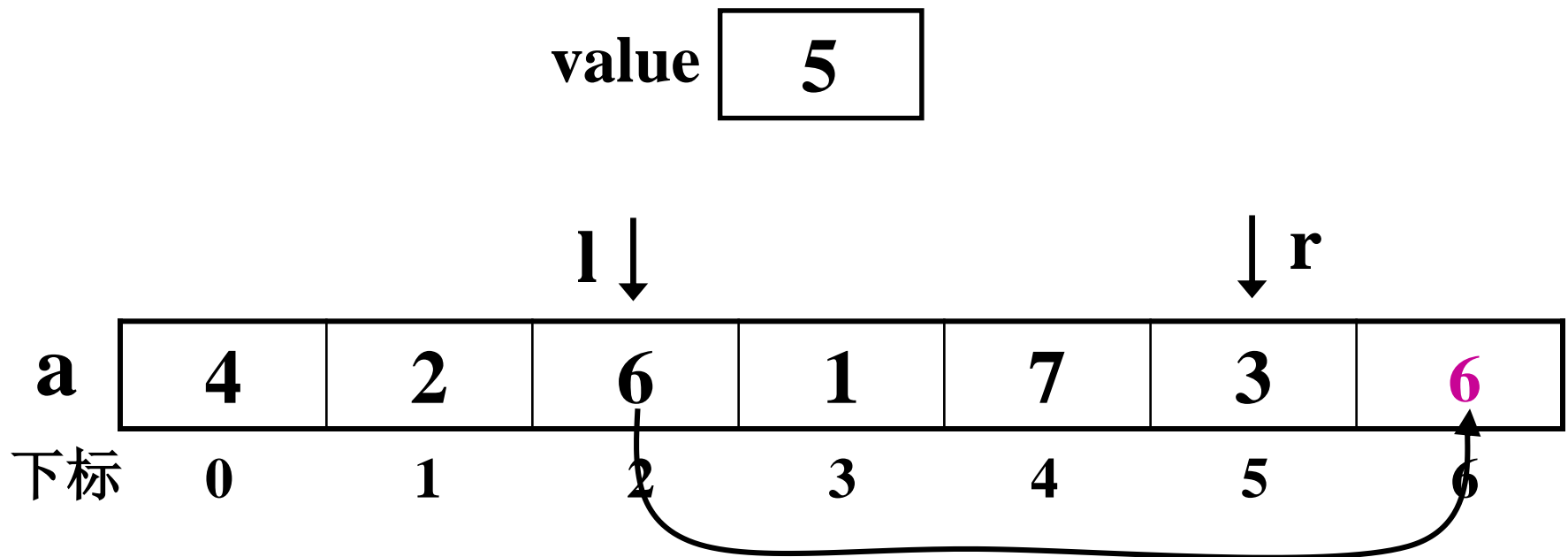
第二步，从 $r=6$ 开始，将 $a[r]$ 与 $value$ 进行比较。
若 $a[r] \geq value$ ，则 $r--$ ，继续比较。否则，
 $a[l] = a[r]$ ， $l++$ 。

value

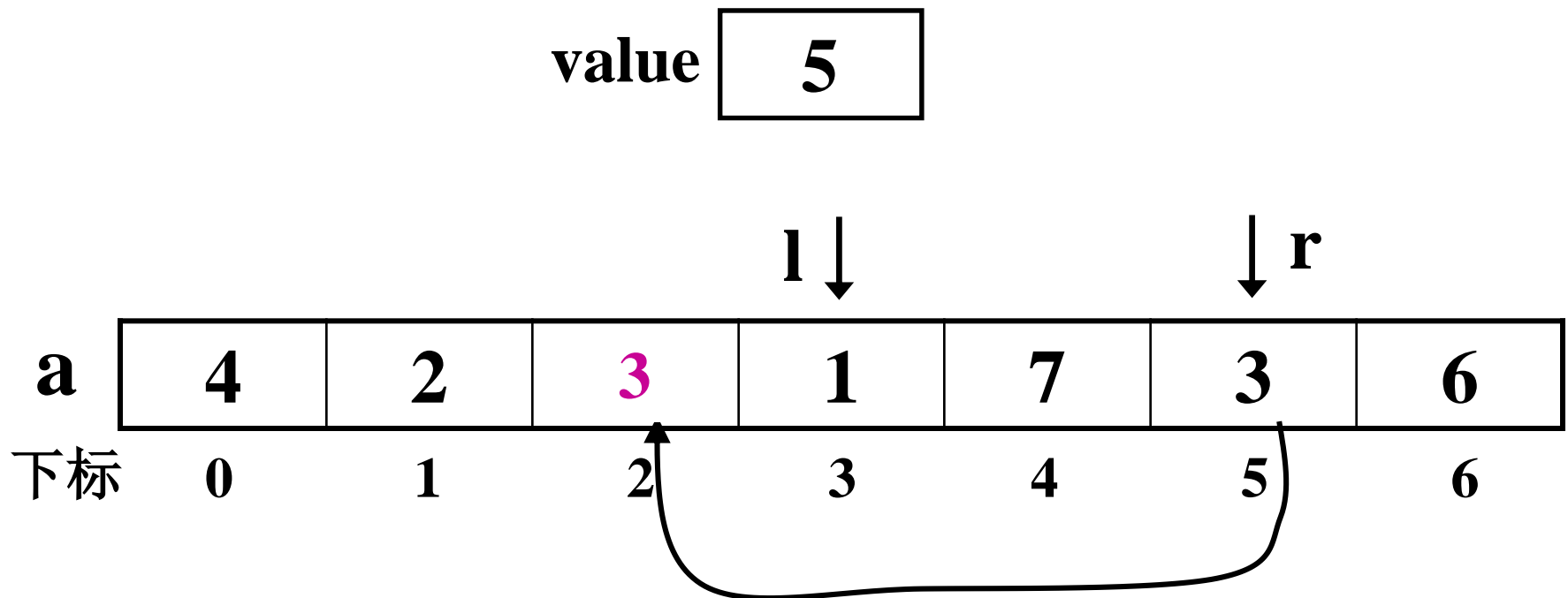
5



第三步，从 $l=1$ 开始，将 $a[l]$ 与 $value$ 进行比较。
若 $a[l] \leq value$ ，则 $l++$ ，继续比较。否则，
 $a[r] = a[l]$ ， $r--$ 。



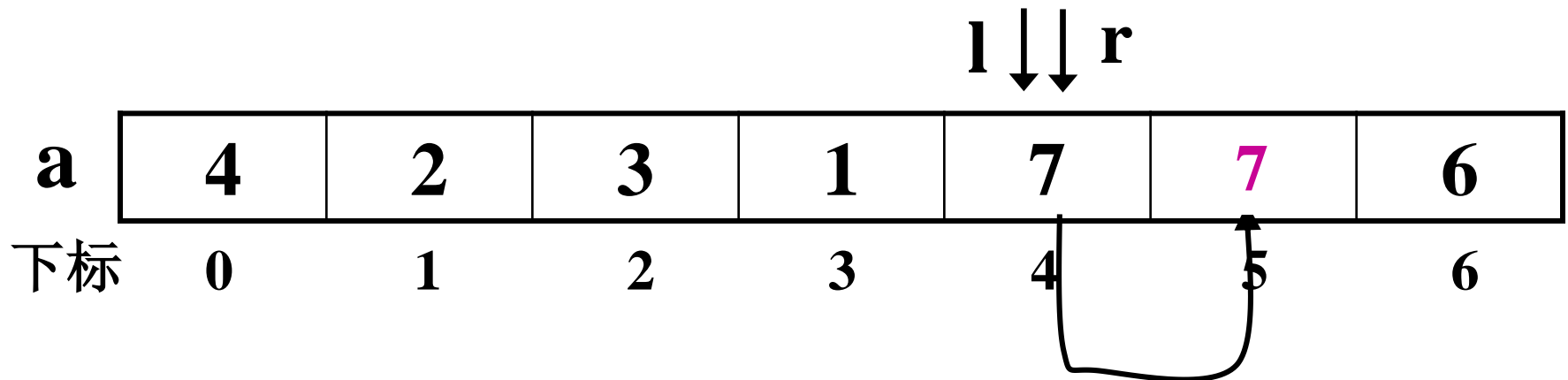
又回到第二步，从 $r=5$ 开始，将 $a[r]$ 与 $value$ 进行比较。若 $a[r] \geq value$ ，则 $r--$ ，继续比较。否则 $a[l] = a[r]$ ， $l++$ 。



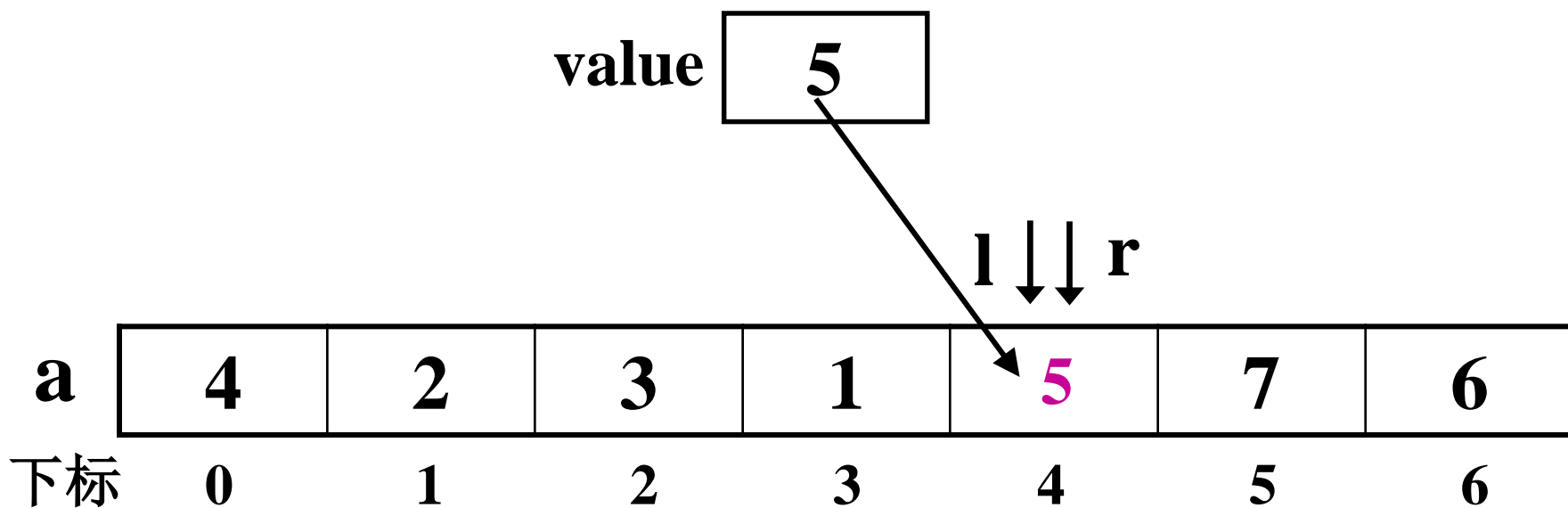
又回到第三步，从 $l=3$ 开始，将 $a[l]$ 与 $value$ 进行比较。若 $a[l] \leq value$ ，则 $l++$ ，继续比较。否则， $a[r] = a[l]$ ， $r--$ 。

value

5



现在 $l = r$ ，已经找到了 **value** 的正确位置，把 **value** 中的值放回到数组当中。



下面的任务：用刚才介绍的方法，对 5 左、右两侧的两段数据，分别进行排序。

a	4	2	3	1	5	7	6
下标	0	1	2	3	4	5	6

l ↓

↓ r

a	4	2	3	1	×	×	×
下标	0	1	2	3	4	5	6

l ↓

↓ r

a	×	×	×	×	×	7	6
下标	0	1	2	3	4	5	6

最后的结果：

a	1	2	3	4	5	6	7
下标	0	1	2	3	4	5	6

具体实现： 几重循环语句？

参考程序： 略.....

```
#include <stdio.h>
#define N 7
void qsort (int a[], int left, int right);
int main()
{
    int a[10], i;
    printf("请输入%d个整数\n", N);
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    qsort(a, 0, N-1);

    for (i = 0; i < N; i++)
        printf("%d ", a[i]);
    return 0;
}
```

```
void qsort (int a[], int left, int right)
```

```
{
```

```
    int l, r, value;
```

```
    if(left >= right)
```

```
        return;
```

```
    l = left; r = right;
```

```
    value = a[l];
```

```
    do
```

```
    {
```

```
        while ((l < r) && (a[r] >= value)) r--;
```

```
        if(l < r) { a[l] = a[r]; l++; }
```

```
        while ((l < r) && a[l] <= value) l++;
```

```
        if(l < r) { a[r] = a[l]; r--; }
```

```
    } while (l != r);
```

```
    a[l] = value;
```

```
    qsort (a, left, l - 1);
```

```
    qsort (a, r + 1, right);
```

```
}
```

请输入7个整数:

5 2 6 1 7 3 4

1 2 3 4 5 6 7

(回顾) 冒泡算法——时间复杂度

- 冒泡排序是一种用时间换空间的排序方法;
- 最坏情况是把顺序的排列变成逆序;
- $(n-1) + (n-2) + \dots + 1 = n * (n - 1) / 2 \approx n^2/2$;
- 时间复杂度: $O(n^2)$;

快排算法——时间复杂度

- 每次将待排序数组分为两个部分，分别处理；
- 理想状况下：每一次都将待排序数组划分成等长两个部分，则需要 $\log n$ 次划分。
- 最坏情况下：数组已经有序或大致有序的情况下，每次划分只能减少一个元素，快速排序将不幸退化为冒泡排序
- 时间复杂度下界 $O(n \log n)$ ，最坏情况 $O(n^2)$

快排算法——时间复杂度

➤ 理想状况下

$$\begin{aligned}T(n) &\leq 2T(n/2) + cn \\&\leq 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\&\dots\dots \\&\leq nT(1) + cn \cdot \log_2 n \approx O(n \log n)\end{aligned}$$

- 快排需要消耗递归栈空间，频繁存取会影响效率
- 常用策略：设置阈值判断

分治法算法——时间复杂度

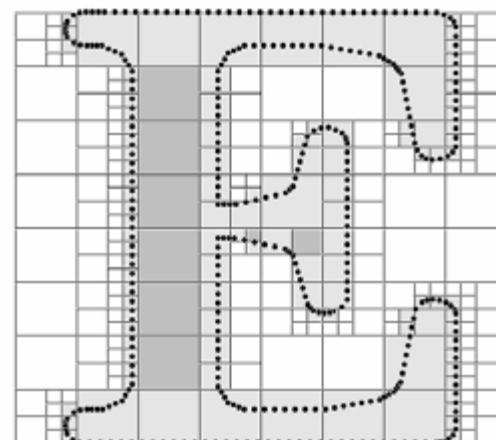
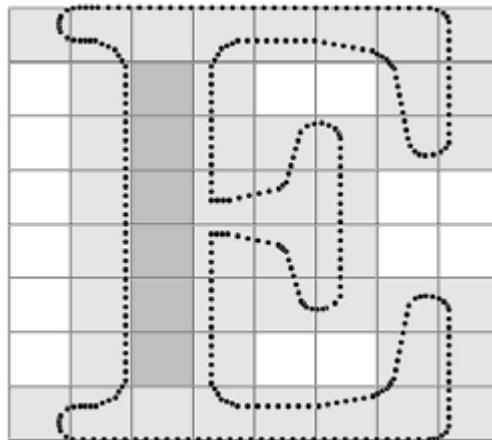
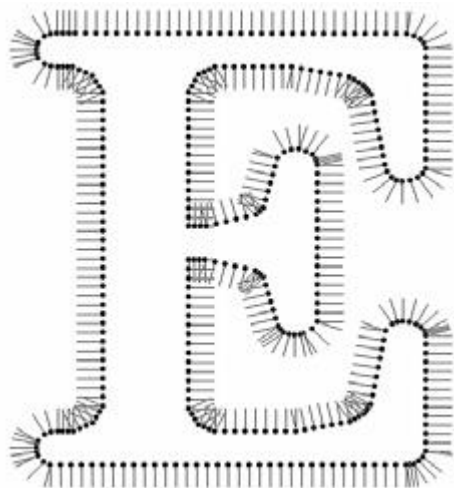
- 能以复杂度 $O(n)$ 的算法将大问题**分解**成两个(或多个)子问题的算法
- 分成 c 个**子问题** (divide)
- 再用 $O(n)$ 时间**合并** (conquer)
- 分治算法复杂度 $O(n\log n)$ (\log 以 c 为底)

程序设计中很多算法都是二分为思想! Very important!

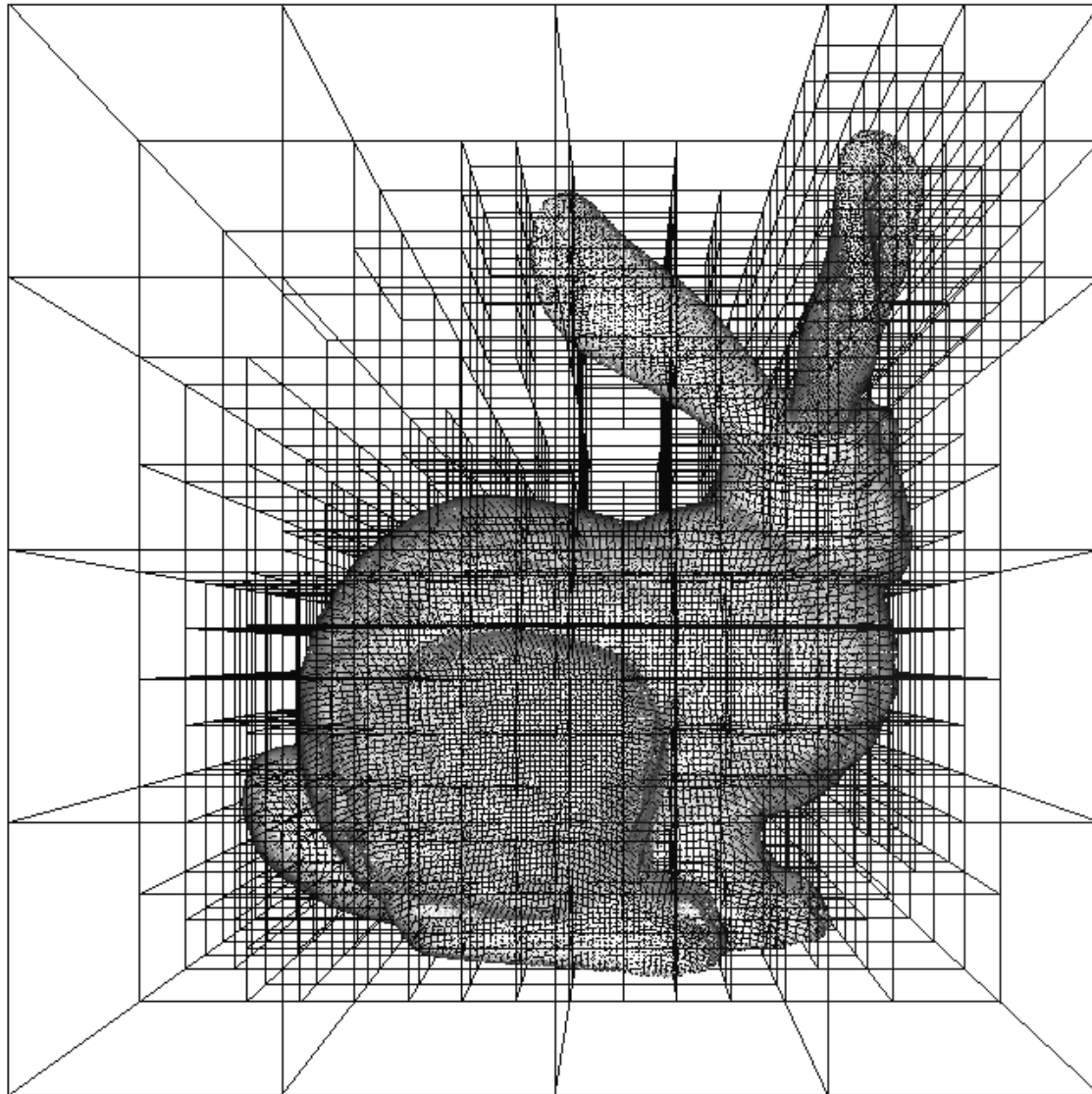
如: 快速排序、二分查找.....

二分能降低时间复杂度, $O(n) \rightarrow O(\log n)$, $O(n^2) \rightarrow O(n\log n)$

四叉树递归分割



八叉树递归分割



```
struct OctTree
```

```
{
```

```
    Node *root;
```

```
};
```

```
struct Node
```

```
{
```

```
    Node* F_NEchild;
```

```
// Front Side
```

```
    Node* F_NWchild;
```

```
    Node* F_SEchild;
```

```
    Node* F_SWchild;
```

```
    Node* B_NEchild;
```

```
// Back Side
```

```
    Node* B_NWchild;
```

```
    Node* B_SEchild;
```

```
    Node* B_SWchild;
```

```
};
```

```
void octree_construction (Node *n, int depth)
{
    if (Node != NULL && depth <= 8)
        return;

    Node* childFNW = new Node(F_NW,n);
    octree_construction (childFNW, depth+1);

    Node* childFNE = new Node(F_NE,n);
    octree_construction (childFNE, depth+1);

    Node* childFSE = new Node(F_SE,n);
    octree_construction (childFSW, depth+1);
    .....
}
```

Lecture 8 - Summary

- **Topics covered:**
 - Recursive programming
 - Divide-and-Conquer algorithms
 - Backtracking algorithms
 - Factorial, Binary Search, Towers of Hanoi, Eight Queens, Quick Sort
 -