第2章 递归与分治策略

- 概述
  - 递归的概念
  - 分治法的设计思想
- 典型应用
  - 快速排序 （2-1）★
  - 归并排序 （2-2）★
  - 二分搜索技术 （2-3）★
  - 线性时间选择 （2-4）★
  - 棋盘覆盖 （2-5）
  - 循环赛日程表 （2-6）★
- 总结

对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决；

否则将其**分解**为k个规模较小的子问题，这些子问题**互相独立**且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。

这种算法设计策略叫做**分治法**。

- 分而治之法的思想
  - 分（**Divide**）： 递归解决若干个较小的问题
  - 治（ **Conquer** ）：从子问题的答案中形成原始问题的解

分治的基本思想:

《孙子兵法》有云：“凡治众如治寡,分数是也 。”

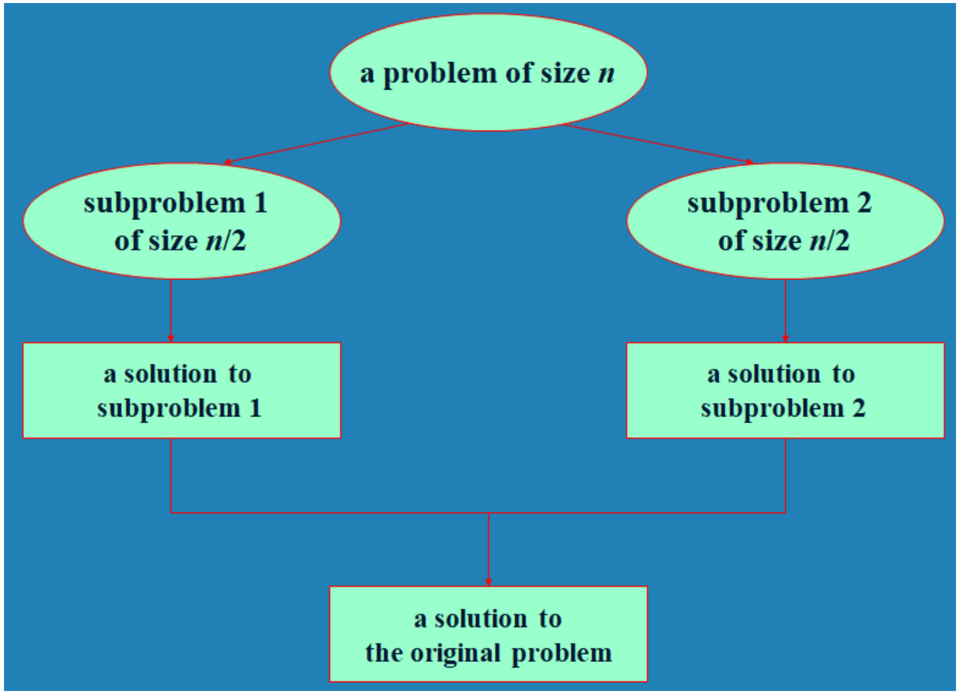
分治法在我们日常生活中也最为常见。
比如管理一个国家:先把国家划分为许多省份，再把每个省份划分为若干个市，依此类推，市===》县（区） ===》乡（镇）===》村。这就是分治。

拆分是降低难度最重要、最有效的方法，人们总会遇到难度大于能力的问题，所以就需要拆分这种方法，把问题难度降低，从而使得能力大于问题难度，把问题解决。

分治算法的总体设计思想就是“分、治、合”。所对应的步骤也是“分、治、合”。

分治法的步骤:

1. 将问题的实例划分为同一个问题的几个较小的实例,最好拥有同样的规模;
2. 对这些较小的实例求解(一般使用递归的方法,但在问题规模足够小的时候,有时也会使用一些其他方法)
3. 如果必要的话,合并这些较小问题的解,以得到原始问题的解.



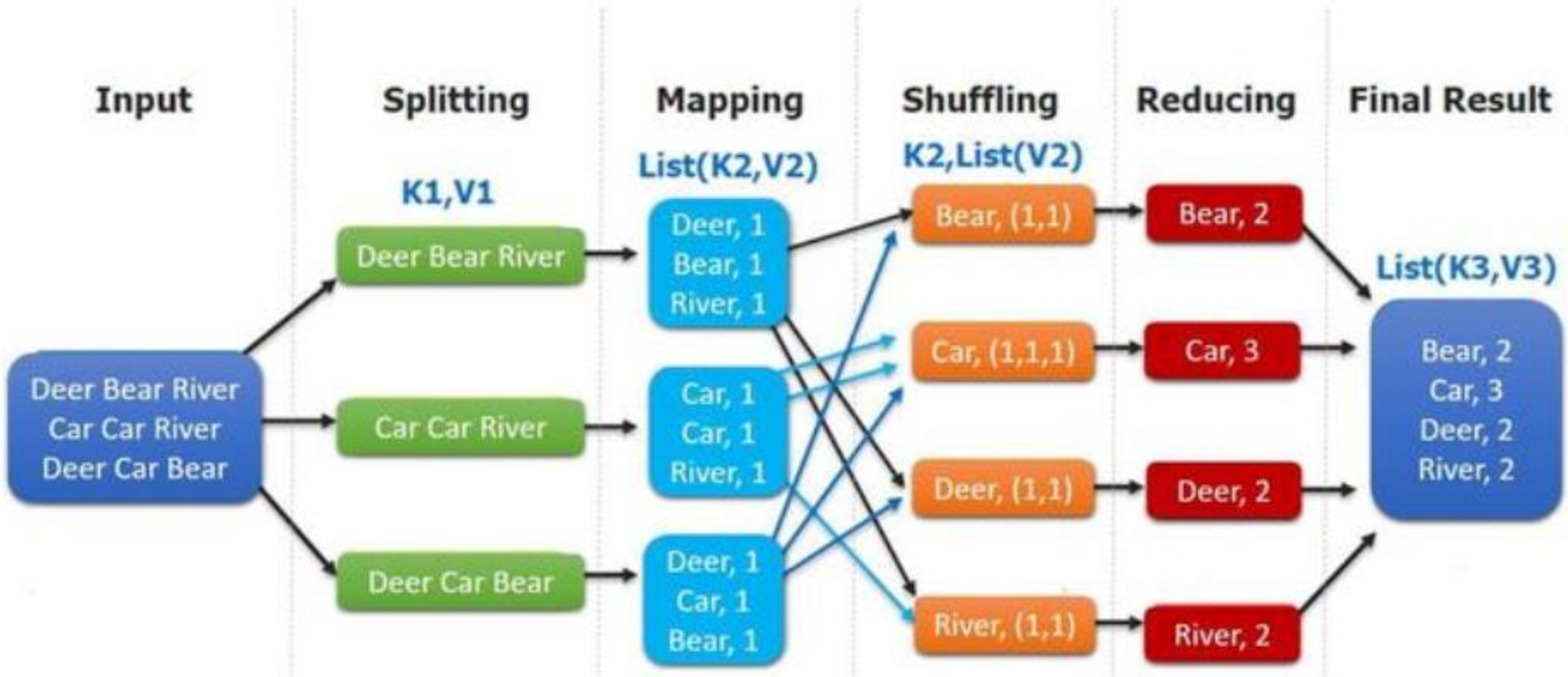
分治方法的应用

赘述。云计算的关键之一是，如何把一个非常大的计算问题，自动分解到许多计算能力不是很强大的计算机上，共同完成。针对这个问题，Google 给出的解决工具是一个叫 MapReduce 的程序，其根本原理就是十分常见的分治算法（Divide-and-Conquer），我称之为“各个击破”法。

这就是 MapReduce 的根本原理。将一个大任务拆分成小的子任务，并且完成子任务的计算，这个过程叫做 Map，将中间结果合并成最终结果，这个过程叫做 Reduce。当然，如何将一个大矩阵自动拆分，保证各个服务器负载均衡，如何合并返回值，就是 MapReduce 在工程上所做的事情了。

MapReduce是一种分布式计算框架，以一种可靠的，具有容错能力的方式并行地处理上TB级别的海量数据集。主要用于搜索领域，解决海量数据的计算问题。

Map负责把一个大的**block**块进行切片并计算。**Reduce**负责把**Map**切片的数据进行汇总、计算。



图示：**MapReduce算法流**

2.1 递归的概念

递归的定义

在定义一个过程或函数时出现调用本过程或本函数的成分，称之为递归。

若调用自身，称之为直接递归。若过程或函数p调用过程或函数q，而q又调用p，称之为间接递归。任何间接递归都可以等价地转换为直接递归，本章主要讨论直接递归。

如果一个递归过程或递归函数中递归调用语句是最后一条执行语句，则称这种递归调用为尾递归。

【例1】设计求$n!$（n为正整数）的递归算法。

解：对应的递归函数如下：

```
int fun(int n)
{   if (n==1)                  //语句1
        return(1);             //语句2
    else                       //语句3
        return(fun(n-1)*n);    //语句4
}
```

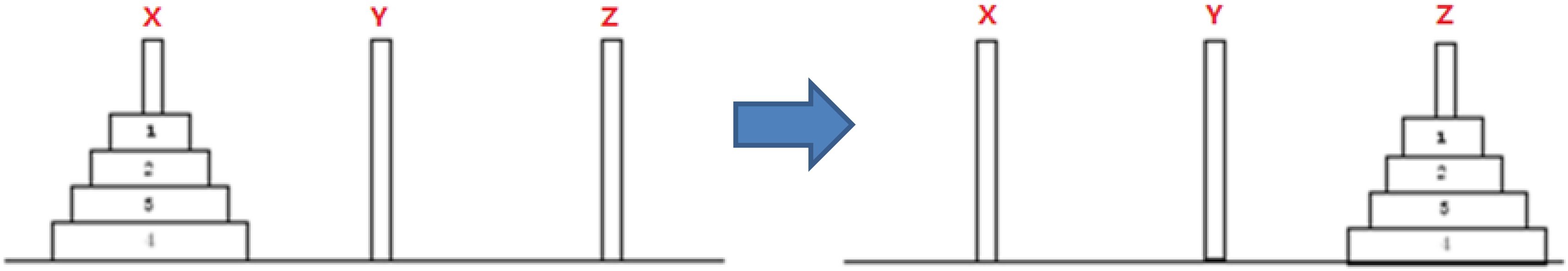
在该函数fun(n)求解过程中，直接调用fun(n-1)（语句4）自身，所以它是一个直接递归函数。

又由于递归调用是最后一条语句，所以它又属于尾递归。

【例2】Hanoi问题

设有3个分别命名为X,Y和Z的塔座，在塔座X上有*n*个直径各不相同，从小到大依次编号为1,2,...,*n*的盘片，现要求将X塔座上的*n*个盘片移到塔座Z上并仍按同样顺序叠放。

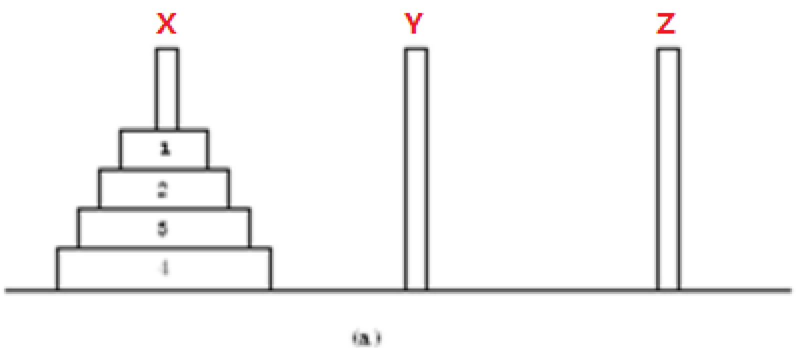
盘片移动时必须遵守以下规则：每次只能移动一个盘片；盘片可以插在X、Y和Z中任一塔座；任何时候都不能将一个较大的盘片放在较小的盘片上。

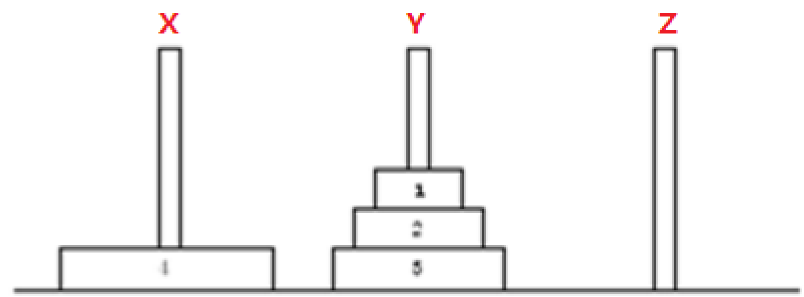


```
Hanoi(n,x,y,z)
```

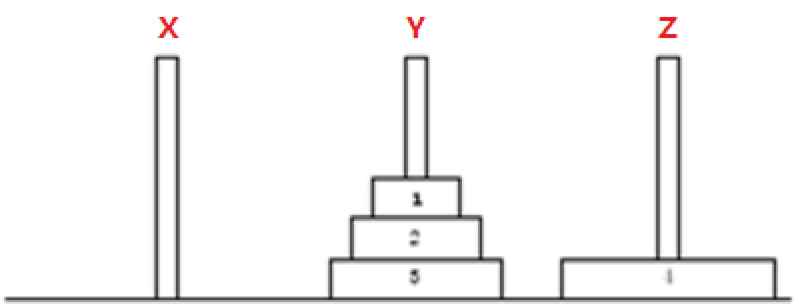
```
Hanoi(n-1,x,z,y);
move(n,x,z):将第n个圆盘从x移到z;
Hanoi(n-1,y,x,z)
```

"大问题"转化为若干个"小问题"求解

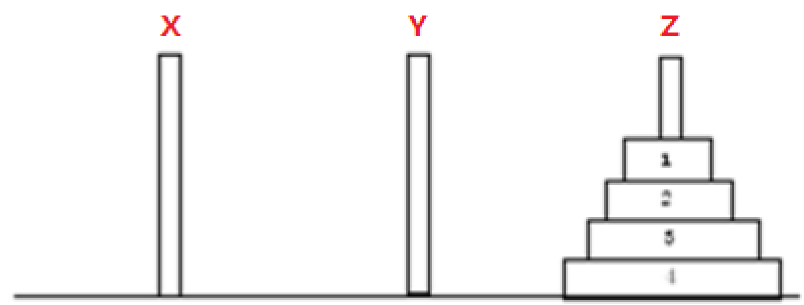




Hanoi(n-1,x,z,y)



move(n,x,z)



Hanoi(n-1,y,x,z)

能够用递归解决的问题 ★

（1）需要解决的问题可以转化为一个或多个子问题来求解，而这些子问题的求解方法与原问题完全相同，只是在数量规模上不同；

（2）递归调用的次数必须是有限的；

（3）必须有结束递归的条件（递归基）来终止递归。

递归算法的执行过程

一个正确的递归程序虽然每次调用的是相同的子程序，但它的参量、输入数据等均有变化，并且在正常的情况下，随着调用的不断深入，必定会出现调用到某一层的函数时，不再执行递归调用而终止函数的执行，遇到递归出口便是这种情况。

递归调用是函数嵌套调用的一种特殊情况，即它是调用自身代码。因此，也可以把每一次递归调用理解成调用自身代码的一个复制件。由于每次调用时，它的参量和局部变量均不相同，因而也就保证了各个复制件执行时的独立性。

但这些调用在内部实现时，并不是每次调用真的去复制一个复制件存放到内存中，而是采用代码共享的方式，也就是它们都是调用同一个函数的代码，而系统为每一次调用开辟一组存储单元，用来存放本次调用的返回地址以及被中断的函数的参量值。

这些单元以栈的形式存放，每调用一次进栈一次，当返回时执行出栈操作，把当前栈顶保留的值送回相应的参量中进行恢复，并按栈顶中的返回地址，从断点继续执行。

求5! 即执行fun(5)时内部栈的变化及求解过程如下：

```
void main()
{  printf("%d\n",fun(5));  }
```

应深刻理解此递归执行的求解过程！

fun(5)调用：进栈 *栈顶*

n	函数值
5	fun(4)*5

fun(4)调用：进栈 *栈顶*

n	函数值
4	fun(3)*4
5	fun(4)*5

fun(3)调用：进栈 *栈顶*

n	函数值
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

n	函数值
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

fun(2)调用：进栈

fun(1)调用：进栈并求值

1	1
2	fun(1)*2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

执行到了递归基

退栈1次并求fun(2)值

2	1*2 = 2
3	fun(2)*3
4	fun(3)*4
5	fun(4)*5

退栈1次并求fun(3)值

3	2*3=6
4	fun(3)*4
5	fun(4)*5

退栈1次并求fun(4)值

4	6*4=24
5	fun(4)*5

退栈1次并求fun(5)值

5	24*5=120

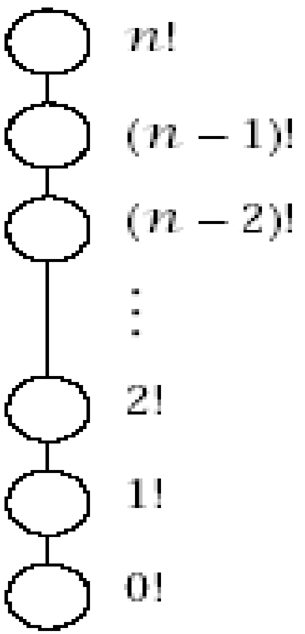
退栈1次并输出120

从以上过程可以得出：

（1）每递归调用一次，就需进栈一次，最多的进栈元素个数称为递归深度，当n越大，递归深度越深，开辟的栈空间也越大。

（2）每当遇到递归出口或完成本次执行时，需退栈一次，并恢复参量值，当全部执行完毕时，栈应为空。

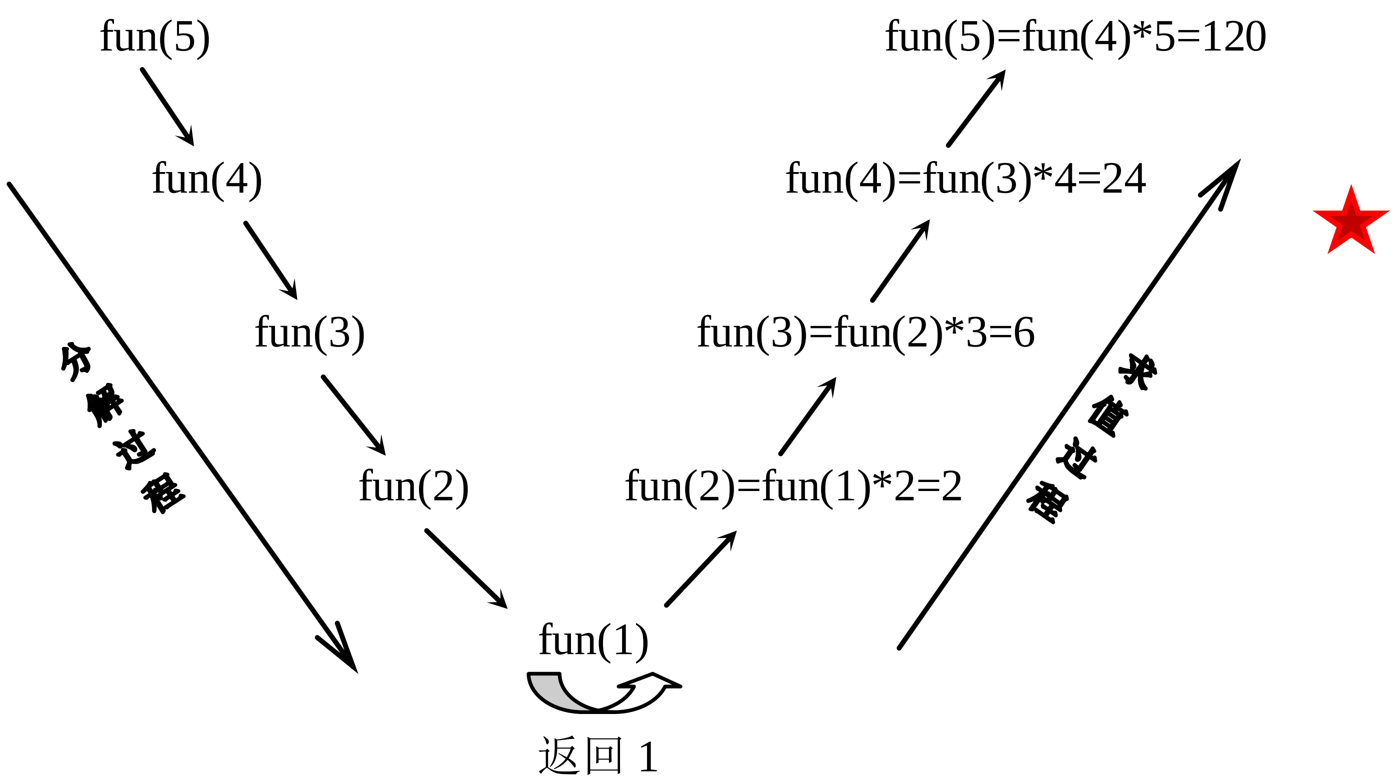
$$
\begin{aligned}
\text{factorial}(5) &= 5 * \text{factorial}(4) \\
&= 5 * (4 * \text{factorial}(3)) \\
&= 5 * (4 * (3 * \text{factorial}(2))) \\
&= 5 * (4 * (3 * (2 * \text{factorial}(1)))) \\
&= 5 * (4 * (3 * (2 * (1 * \text{factorial}(0))))) \\
&= 5 * (4 * (3 * (2 * (1 * 1)))) \\
&= 5 * (4 * (3 * (2 * 1))) \\
&= 5 * (4 * (3 * 6)) \\
&= 5 * (4 * 6) \\
&= 5 * 24 \\
&= 120.
\end{aligned}
$$



归纳起来，递归调用的实现是分两步进行的：

第一步是分解过程，即用递归体将“大问题”分解成“小问题”，直到递归出口（递归基）为止，然后进行

第二步的求值过程，即已知“小问题”，计算“大问题”。前面的**fun(5)**求解过程如下图所示。



2.2 分治法的设计思想

分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以容易地解决。

（2）该问题可以分解为若干个规模较小的相同问题。

（3）利用该问题分解出的子问题的解可以合并为该问题的解。

（4）该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

凡治众如治寡，分数是也。

----孙子兵法

分治法的求解过程

分治法通常采用递归算法设计技术，在每一层递归上都有3个步骤：

①分解	将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题。
② 求解子问题	若子问题规模较小而容易被解决则直接求解，否则递归地求解各个子问题。
③ 合并	将各个子问题的解合并为原问题的解。

分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法

★分治法的一般的算法设计模式如下：

```
divide-and-conquer(P)
{
    if |P|≤n0 return adhoc(P);        // 返回递归基

    将P分解为较小的子问题 P1, P2, …, Pk;
    for(i=1;i<=k;i++)                 //循环处理k次
        yi=divide-and-conquer(Pi);    //递归解决Pi

    return merge(y1,y2,…,yk);         //合并子问题
}
```

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？

这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的*k*个子问题的处理方法是行之有效的。

减治法（Decrease and conquer）

k=1时，成为减治法，为求解一个大规模的问题，可以将其缩减为一个更小的问题。

【例】计算任意n个整数之和：

```
int sum(int a[], int n)
{
    return (n<1)? 0 : sum(a, n-1)+a[n-1];
}
```

- 从递归角度看，为求解sum(a, n)，需要递归求解规模为n-1的问题sum(a, n-1)，再累加上a[n-1] 。
- 递归基：sum(a, 0)
- 递归方程：T(n) = T(n-1) + O(1)。

  T(0) = O(1)

分治法（Divide and Conquer）

K>=2时，成为分治法，为求解一个大规模的问题，可以将其缩减为一个若干个（通常两个）更小的问题，规模大体相当，分别求解子问题。子问题归并后，获得原问题的解。

许多问题可以取 *k*=2，称为二分法，如图所示

求解排序问题

2-1 快速排序★

霍尔(Hoare)

问题描述：

输入：

包含n个数字的无序序列

$< a_0, a_1, a_2, \ldots, a_{n-1} >$；

输出：

$< a_0', a_1', a_2', \ldots, a_{n-1}' >$；

满足

$a_0' <= a_1' <= a_2' <= \ldots <= a_{n-1}'$

于1934 年出生，英国计算机科学家，是著名的快速排序算法的发明者。

- 霍尔本人被称为“影响算法世界的十位大师之一”
- 由霍尔发明的快速排序算法被称为“二十世纪十大算法之一”
- 霍尔领导了Algol 60第一个商用编译器的设计与开发，由于其出色的成绩，最终成为该公司首席科学家
- 因霍尔对Algol 60程序设计语言理论、互动式系统及APL的贡献，1980年被美国计算机协会授予“图灵奖”
- 2000年因为其在计算机科学与教育上做出的贡献被封为爵士

基本思想：

霍尔1961年提出

（1）分解(divide)。S = SL + SR，max（SL）< min（SR）

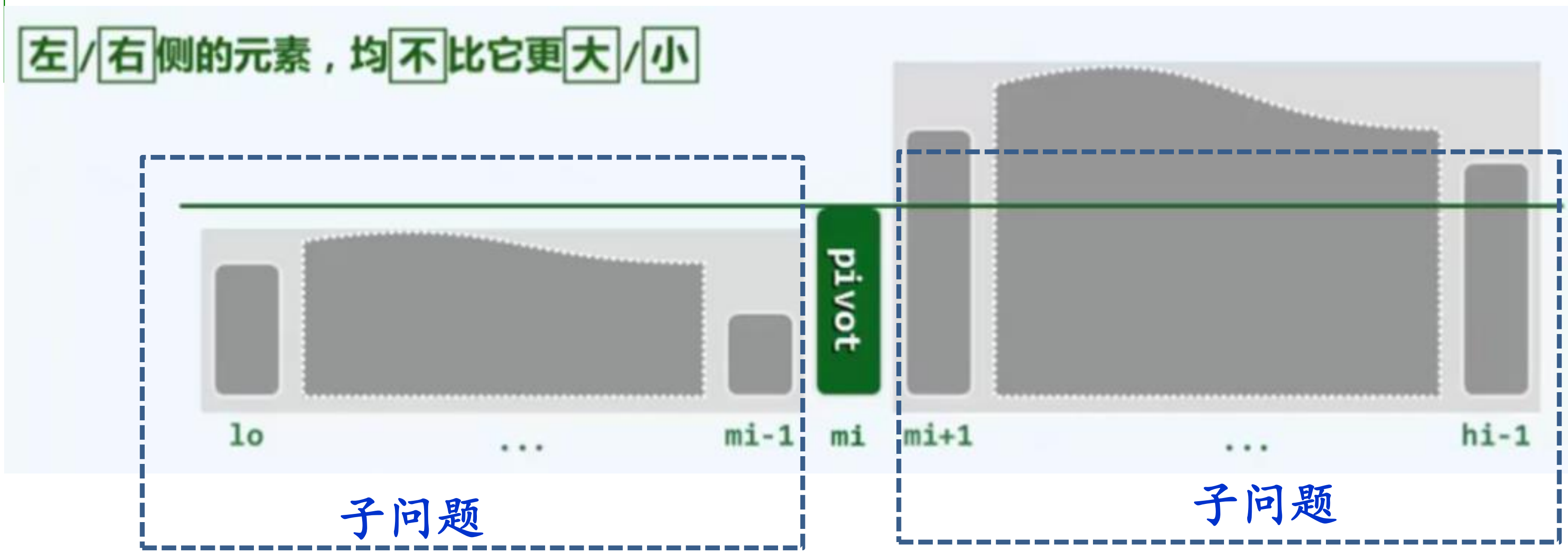
将序列分解为{左边区域、轴点（基准点）、右边区域}

（2）递归求解（conquer）。

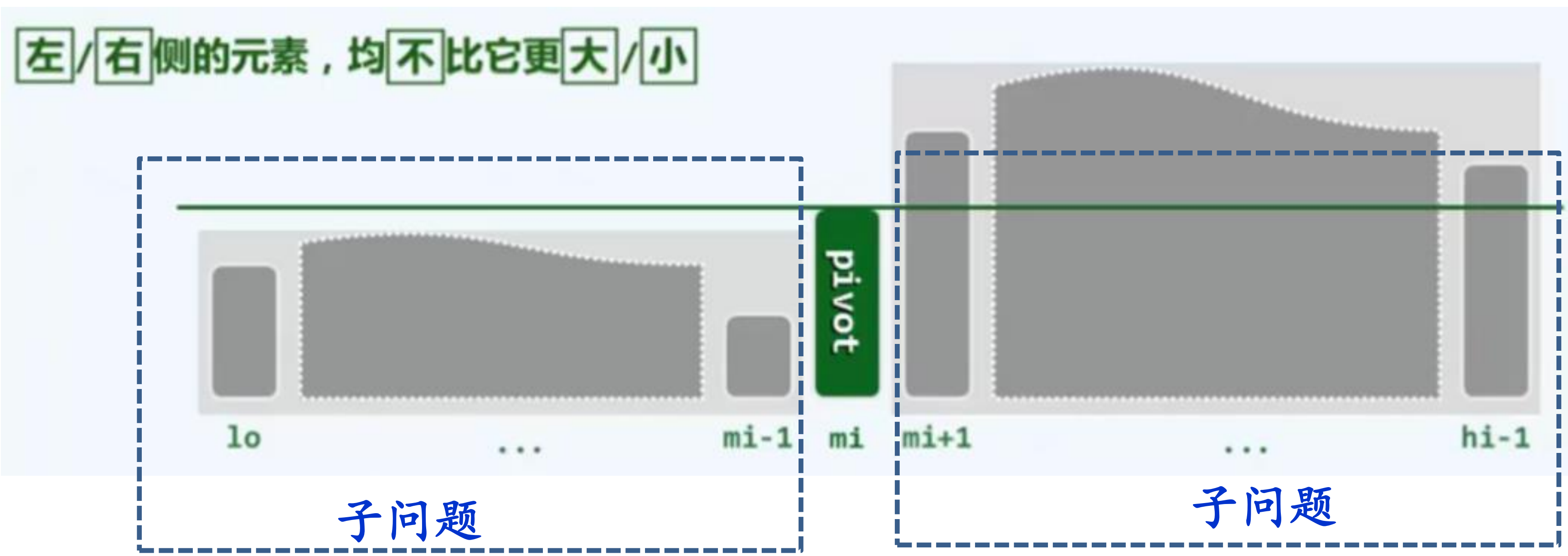
子序列分别递归排序后，原序列自然有序：

sorted(S) = sorted(SL) + sorted(SR)

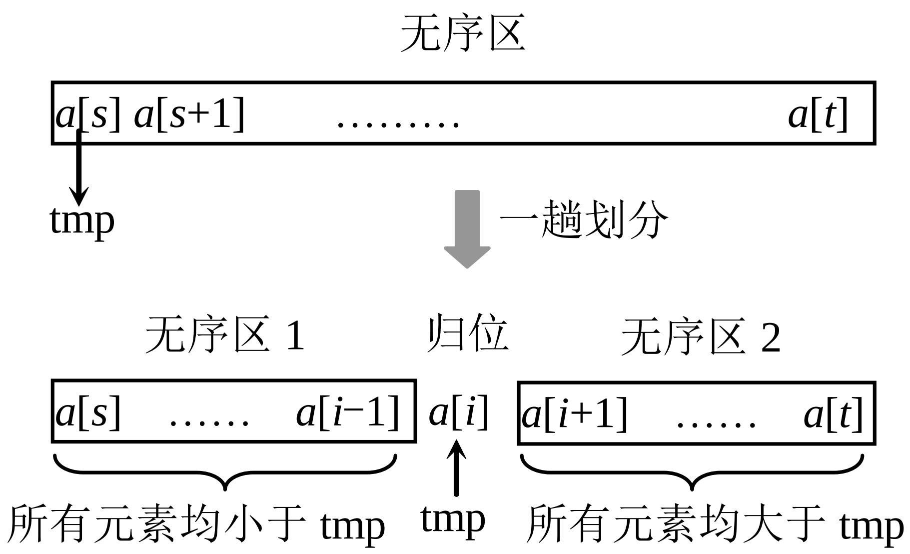
（3）平凡解：只剩单个元素时，该元素本身就是解。



```
void qSort(int  lo,   int hi)        //对下标位于lo~hi区间的元素进行排序
{
     if (lo<hi) {                     //序列内至少存在2个元素的情况
       int  mi=partition(lo, hi);     //划分。mi为基准点位轴点(pivot)的下标
       qSort (lo, mi-1);              //递归对左半段排序。
       qSort (mi+1, hi);              //递归对右半段排序。
       }
 }
```

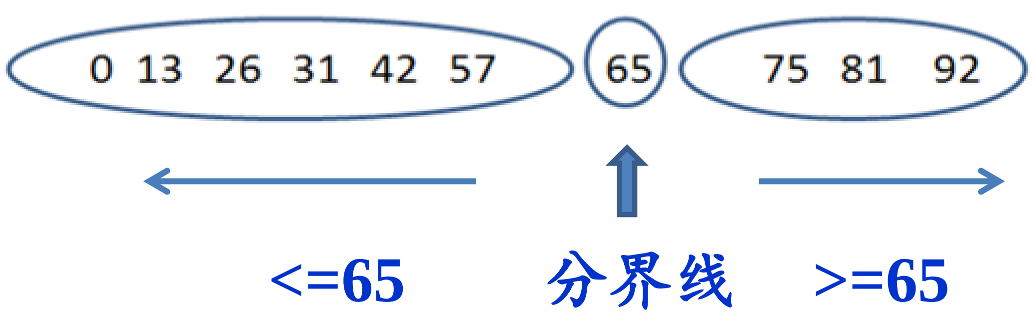


有序队列的每一个数都是轴点

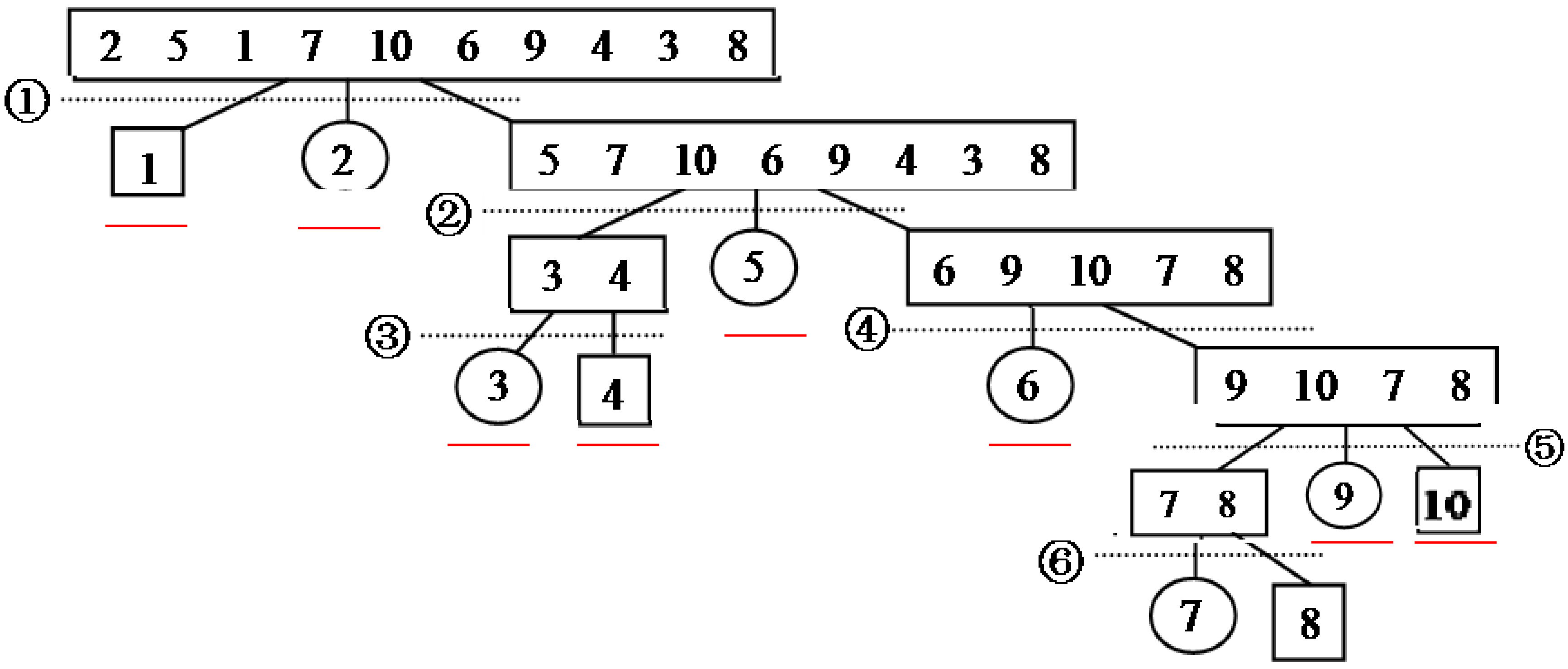


这是一种二分法思想，每次将整个无序序列一分为二，归位一个元素；

对两个子序列采用同样的方式进行排序，直至子序列长度为1或0为止。



例如，对于{2,5,1,7,10,6,9,4,3,8}序列，其快速排序过程如下图所示，图中虚线表示一次划分，虚线旁的数字表示执行次序，圆圈表示归位的基准（轴点）。

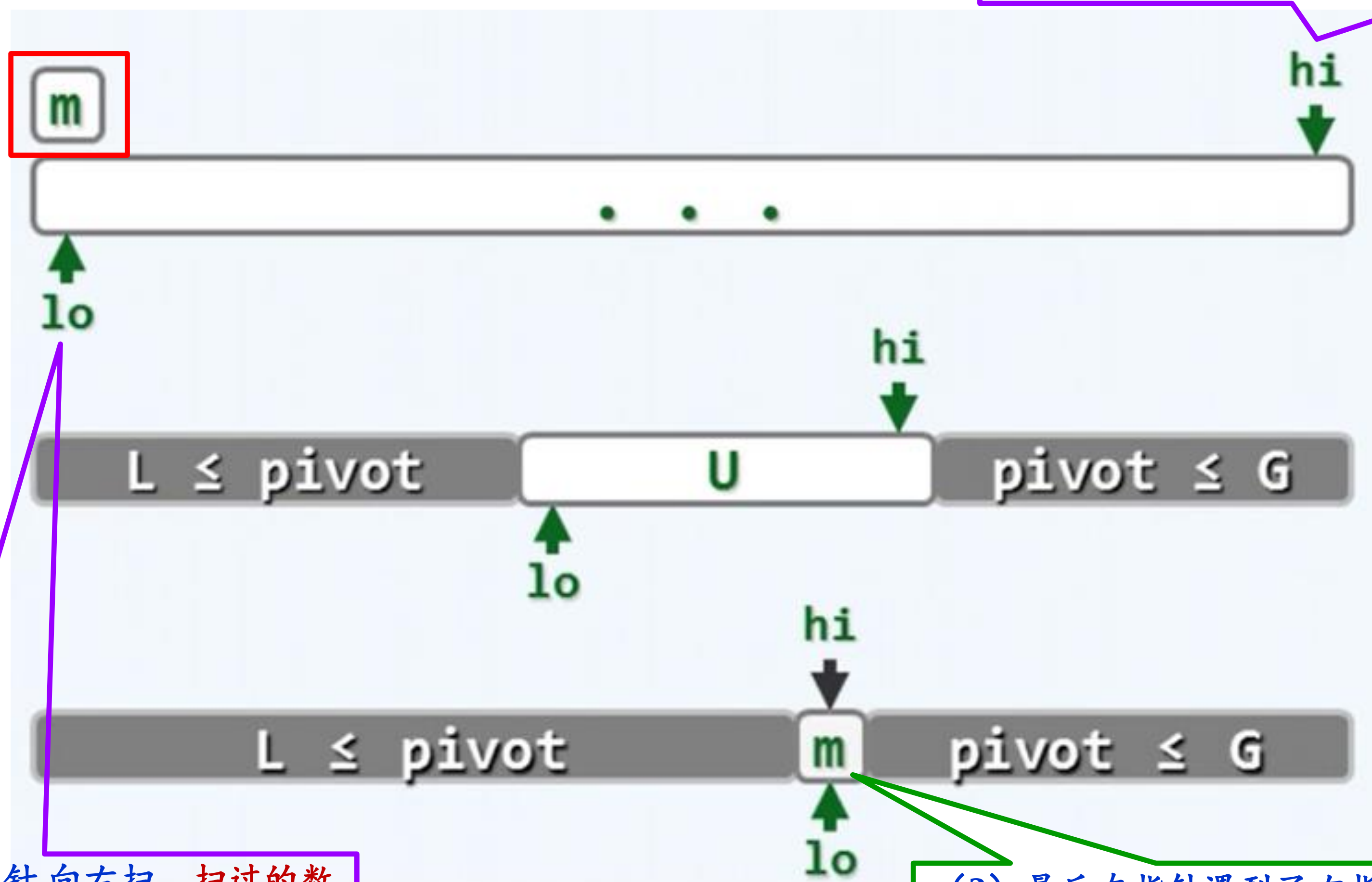


快速排序：就是将每个元素逐个转换为轴点的过程

通过适当交换，使得任一元素转换为轴点！

轴点转换思路：

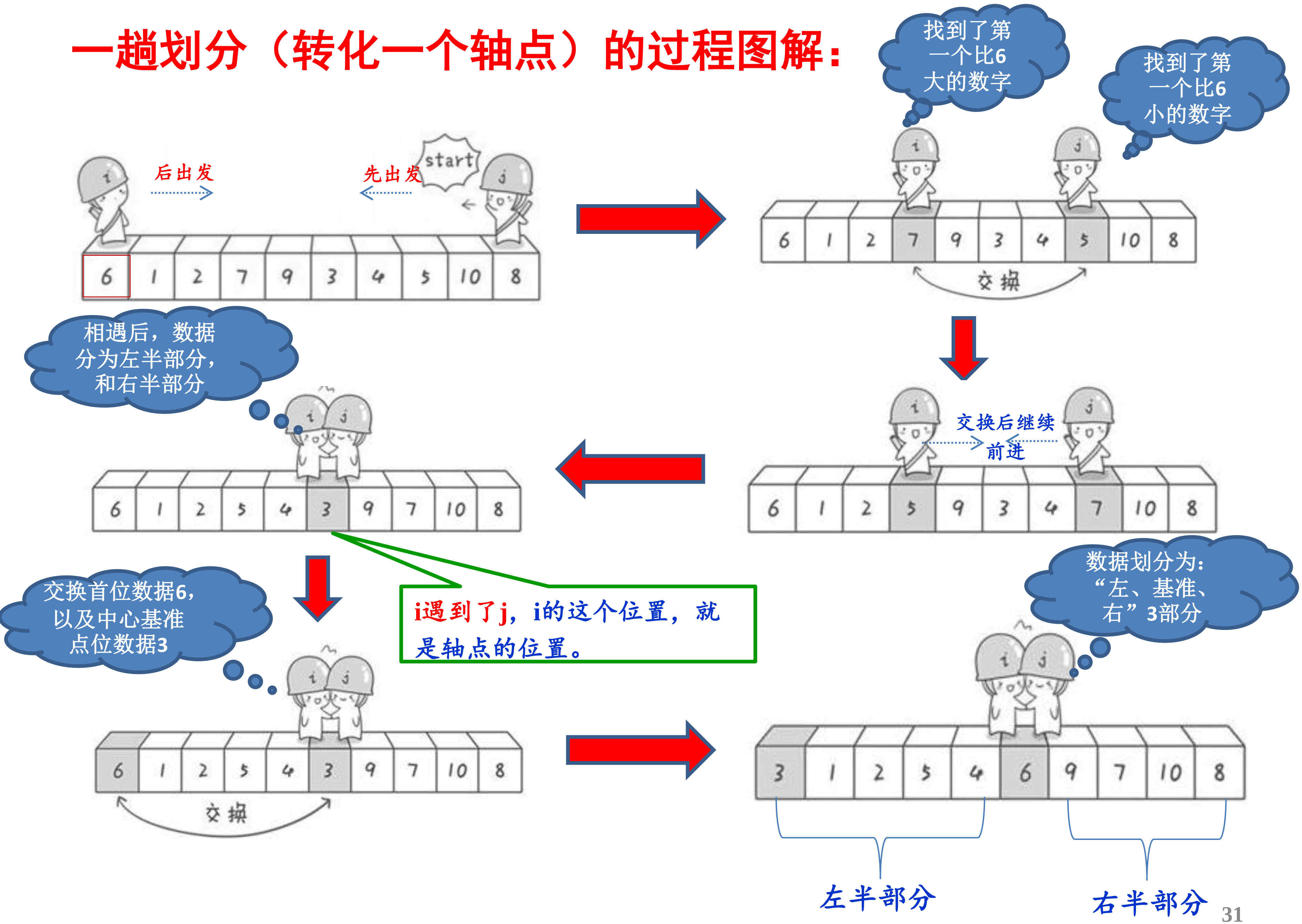
（1）右指针 向左扫，扫过的数都应该比基准数(m)大或者相等，如果扫到了比m小的数，那么这个数应该换到左面去。



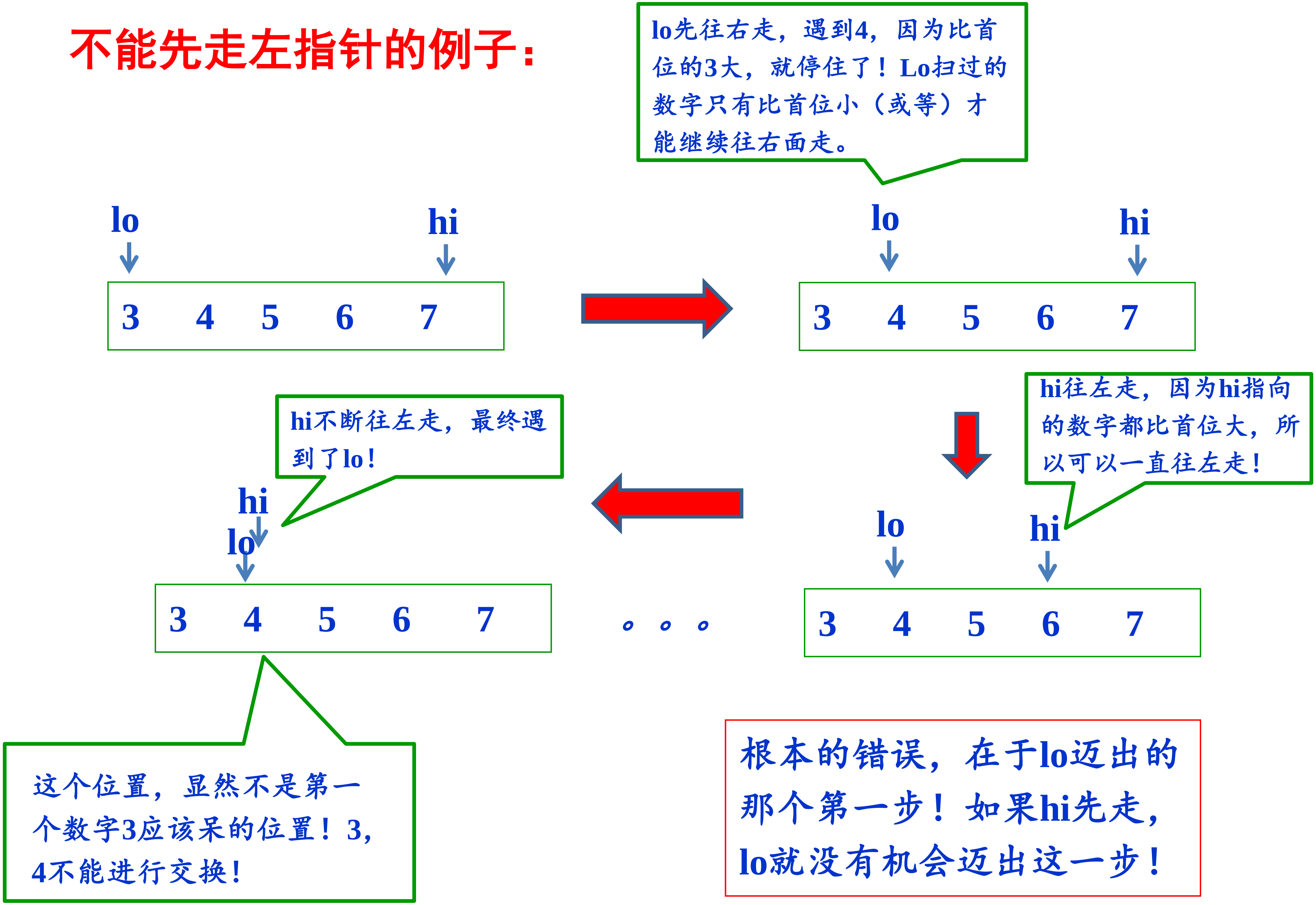
（2）左指针 向右扫，扫过的数都应该比基准数(m)小或者相等，如果扫到了比m大的数，那么这个数应该换到右面去。

（3）最后左指针遇到了右指针，左指针指向的这个位置就是轴点！因为，这个位置右面的数都 >= m，而左面的数都 <= m!

一趟划分（转化一个轴点）的过程图解：

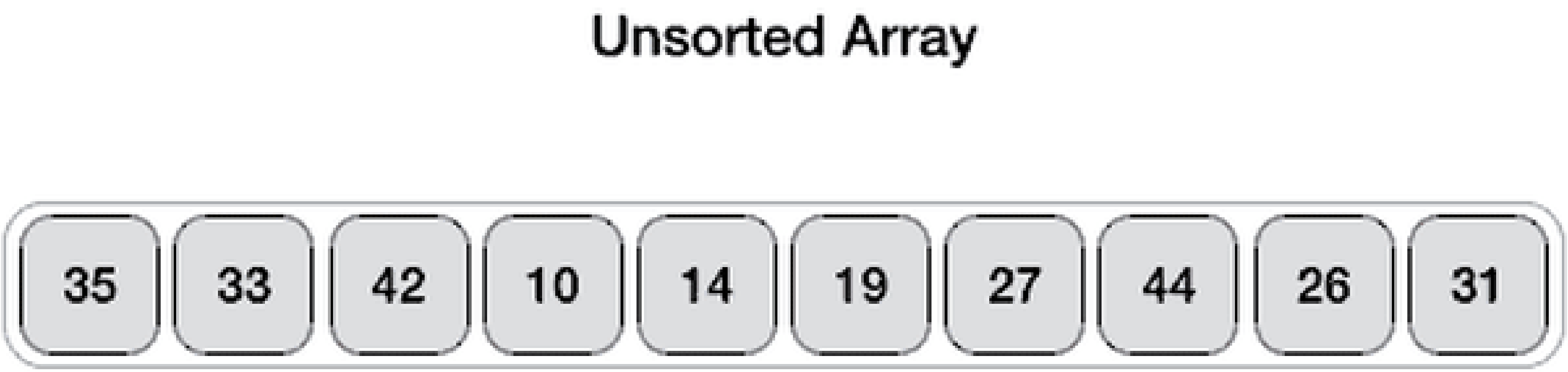


不能先走左指针的例子：



根本的错误，在于lo迈出的那个第一步！如果hi先走，lo就没有机会迈出这一步！

快速排序动画：



参考链接：
https://leetcode-cn.com/problems/sort-an-array/solution/shi-er-chong-pai-xu-suan-fa-bao-ni-man-yi-dai-gift/

快速排序的一趟划分算法程序：

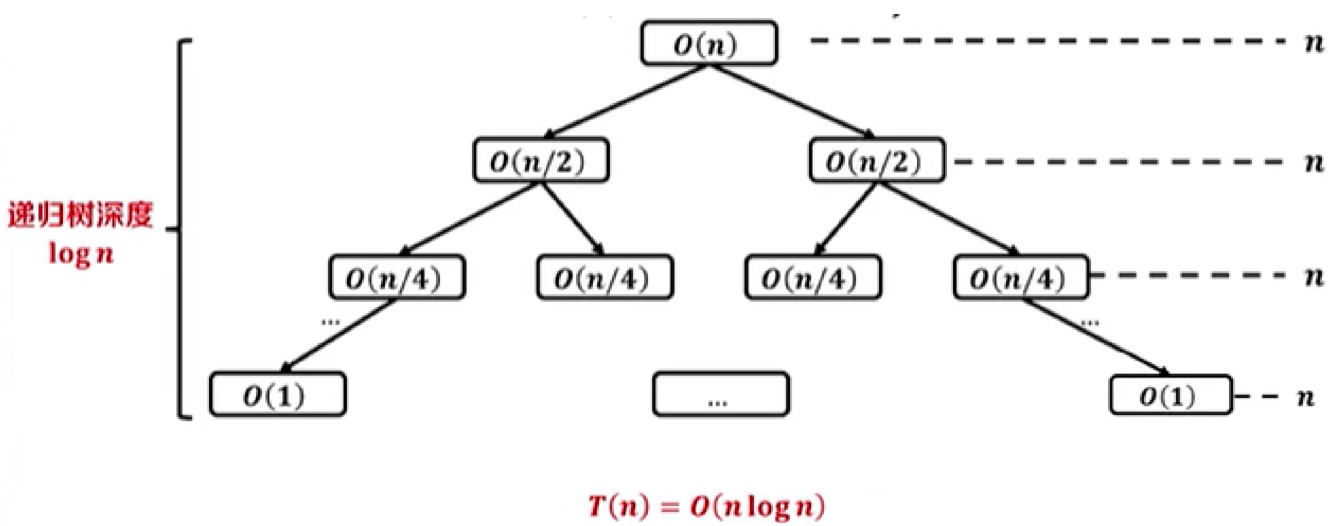
```
// 划分并返回基准点位（轴点）的下标
// a:  待划分数组，执行后为划分好的数组（输入 && 输出）
// lo: 数据区间低位下标； hi: 数据区间高位下标；  （输入）
// 返回：基准点位（轴点）的下标
int Partition(int a[],int lo,int hi)     1 2 3 4 5 6
{
   int i=lo, j=hi;          //左右指针初始化
   int pivot = a[lo];       //用序列的第1个记录作为基准

   //---- 根据基准点进行划分，定位中间位置i  ----
   while (i!=j)           //从序列两端交替向中间扫描,直至i=j为止
   {
      while (i<j && a[j]>= pivot)
          j--;            //从右向左扫描,找第1个关键字小于=pivot的a[j]
      while (i<j && a[i]<= pivot)
          i++;
                          //从基点向右扫描,找第1个关键字大于=pivot的a[i]

      swap(a,i,j);       //对换i, j指向的数字，使得左小右大
   }

  //这时候，a[i]左面都比基准数小或相等，a[i]右面的都比基准位大或相等
  //----让原处于首位的基准（轴）位归位，与a[i]互相交换 ----
   swap(a,lo,i);

   //----  返回基准点位的下标 ----
   return i;
}
```

算法分析1：快速排序的时间主要耗费在划分操作上，对长度为n的区间进行划分，共需n-1次关键字的比较，时间复杂度为O(n)。

对n个记录进行快速排序的过程构成一棵递归树，在这样的递归树中，每一层至多对n个记录进行划分，所花时间为O(n)。**总共时间：**
T(n)=O(n)+2*T(2/n)=**O(n*logn)**

当初始排序数据正序或反序时，此时的递归树高度为n，快速排序呈现最坏情况，即最坏情况下的时间复杂度为O(n^2)：T(n)=O(n)+T(n-1)+T(0)=**O(n*n)**

当初始排序数据随机分布，使每次分成的两个子区间中的记录个数大致相等，此时的递归树高度为$\log_2 n$，快速排序呈现最好情况，即最好情况下的时间复杂度为O($n\log_2 n$)。快速排序算法的平均时间复杂度也是O($n\log_2 n$)。



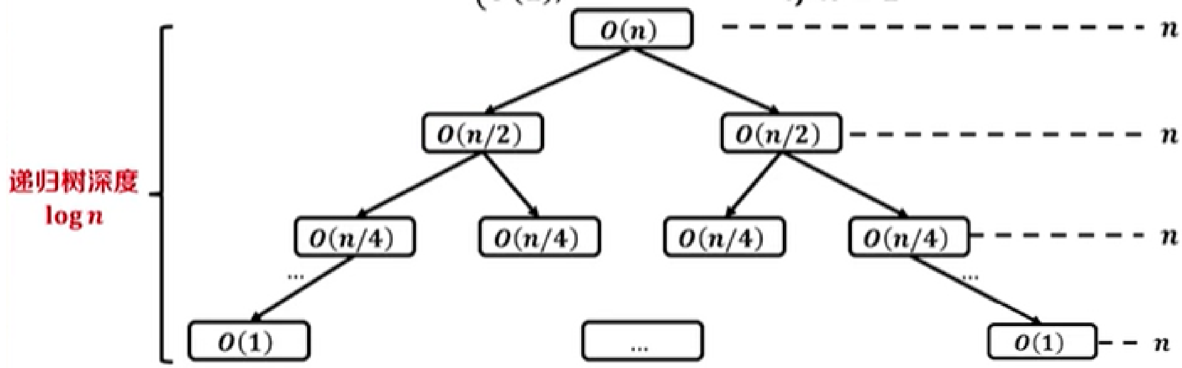
算法分析2： 从快速排序算法的递归树可知，快速排序的趟数取决于递归树的深度。

如果每次划分对一个对象定位后，该对象的左侧子序列与右侧子序列的长度相同，则下一步将是对两个长度减半的子序列进行排序，这是最理想的情况。

在n个元素的序列中，对一个对象定位所需时间为 $O(n)$。若设 $t(n)$ 是对 n 个元素的序列进行排序所需的时间，而且每次对一个对象正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：

$$
\begin{aligned}
T(n) &\le cn + 2\,t(n/2) \qquad // \ c\text{是一个常数} \\
&\le cn + 2\,(\,cn/2 + 2t(n/4)\,) = 2cn + 4t(n/4) \\
&\le 2cn + 4\,(\,cn/4 + 2t(n/8)\,) = 3cn + 8t(n/8) \\
&\ \ \ldots\ldots\ldots \\
&\le cn\log_2 n + nt(1) = o(n\log_2 n)
\end{aligned}
$$

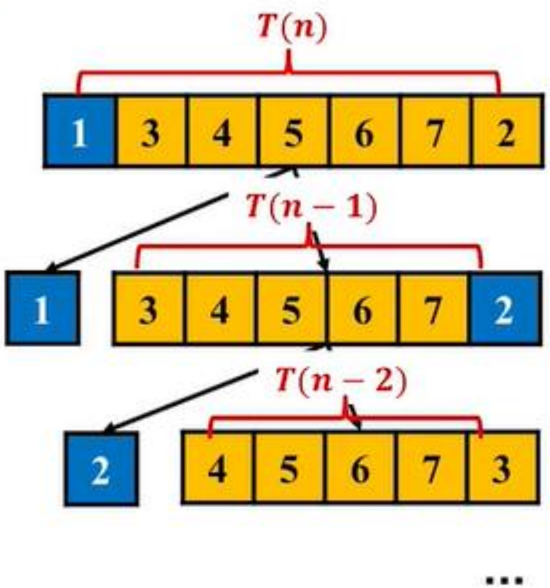
$$
T(n) = \begin{cases} 2T(n/2) + O(n), & if\ n > 1 \\ O(1), & if\ n = 1 \end{cases}
$$



$T(n) = O(n\log n)$

- 最坏时间复杂度：$O(n^2)$
- 平均时间复杂度：$O(n\log n)$
- 辅助空间：$O(n)$或$O(\log n)$
- 稳定性：不稳定

- 最坏情况
  - 数组划分后，每次主元都在一侧



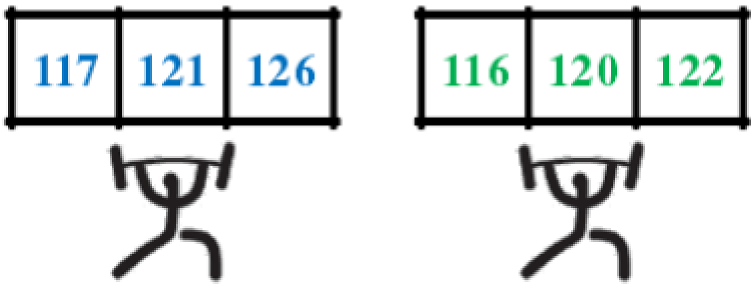
$$T(n) = T(n-1) + T(0) + O(n) = O(n^2)$$

快速排序算法的性能取决于划分的对称性。通过修改算法partition，可以设计出采用随机选择策略的快速排序算法。在快速排序算法的每一步中，当数组还没有被划分时，可以在a[p:r]中**随机选出一个元素作为划分基准**，这样可以使划分基准的选择是随机的，从而可以期望划分是较对称的。

2-2 合并排序（归并排序 merge sort）★

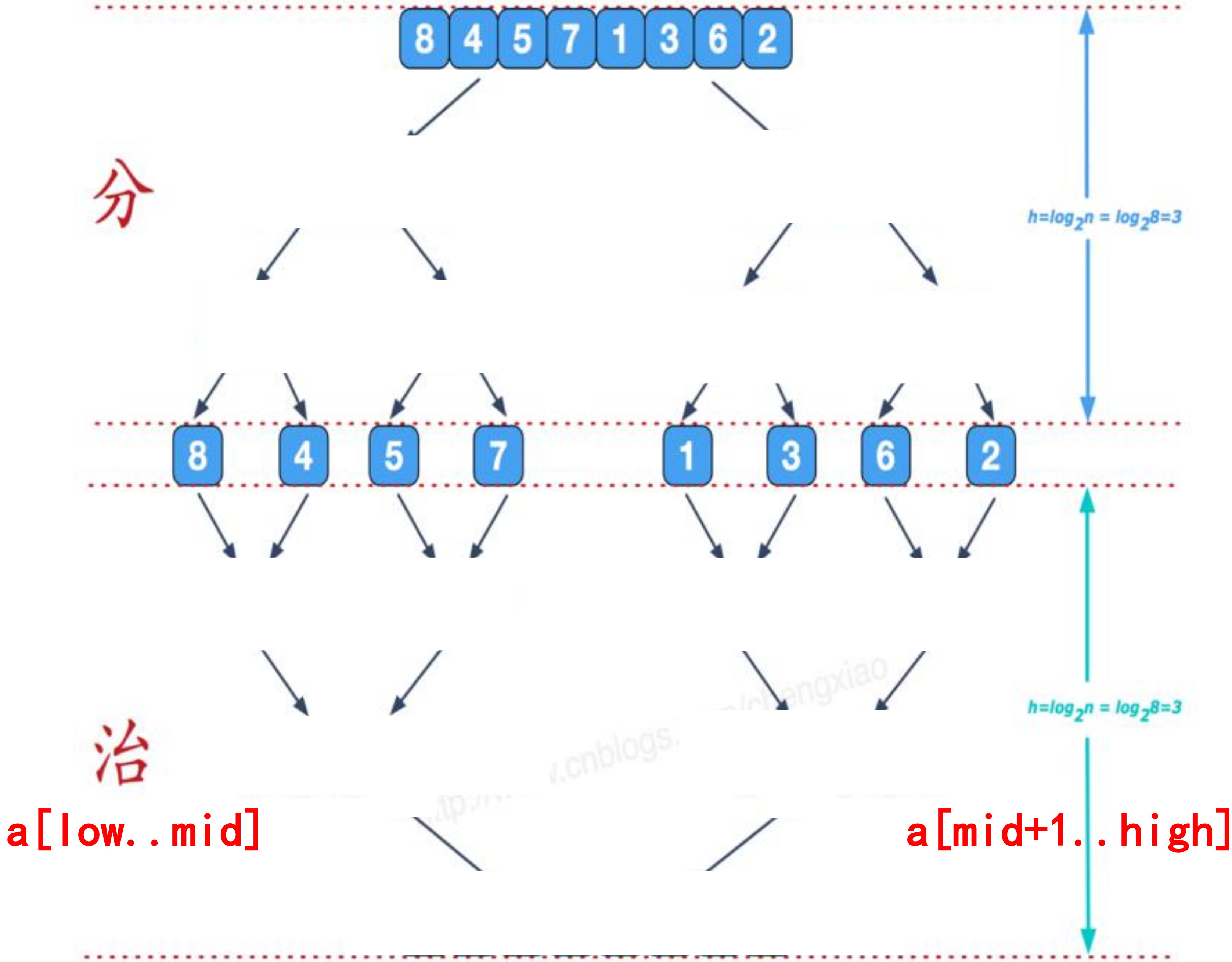
基本思想，利用分治策略

（1）序列一份为二　　　// O(1)
（2）子序列递归排序；　// 2T(n/2)
（3）合并有序的子序列；// O(n)



December 28, 1903 – February 8, 1957

1945年提出


8 4 5 7 1 3 6 2
分
h=log2n = log28=3
8 4 5 7 1 3 6 2
治
h=log2n = log28=3
a[low..mid]
a[mid+1..high]

归并排序动画：

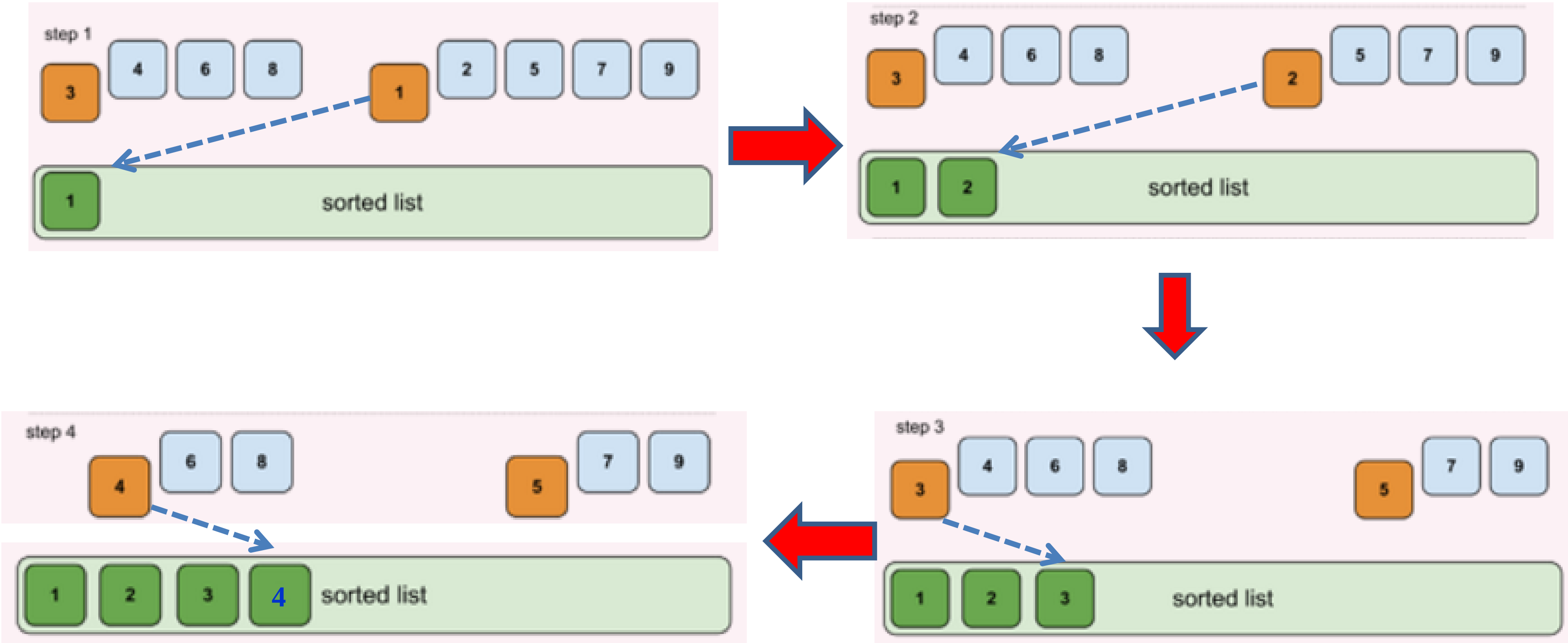
https://leetcode-cn.com/problems/sort-an-array/solution/shi-er-chong-pai-xu-suan-fa-bao-ni-man-yi-dai-gift/

十二种排序算法包你满意（带GIF图解）

https://leetcode-cn.com/problems/sort-an-array/solution/dong-hua-mo-ni-yi-ge-kuai-su-pai-xu-wo-x-7n7g/

【动画模拟】快排

归并过程的图解：


step 1
3
4
6
8
1
2
5
7
9
1
sorted list
step 2
3
4
6
8
2
5
7
9
1
2
sorted list
step 3
3
4
6
8
5
7
9
1
2
3
sorted list
step 4
4
6
8
5
7
9
1
2
3
4
sorted list

归并操作的工作原理如下:

(1) 第一步：申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列;

(2) 第二步：设定两个指针，最初位置分别为两个已经排序序列的起始位置;

(3) 第三步：比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置;

(4) 重复步骤3直到某一指针超出序列尾;

(5) 倒数第二步：将剩下序列剩下的所有元素直接复制到合并序列尾;

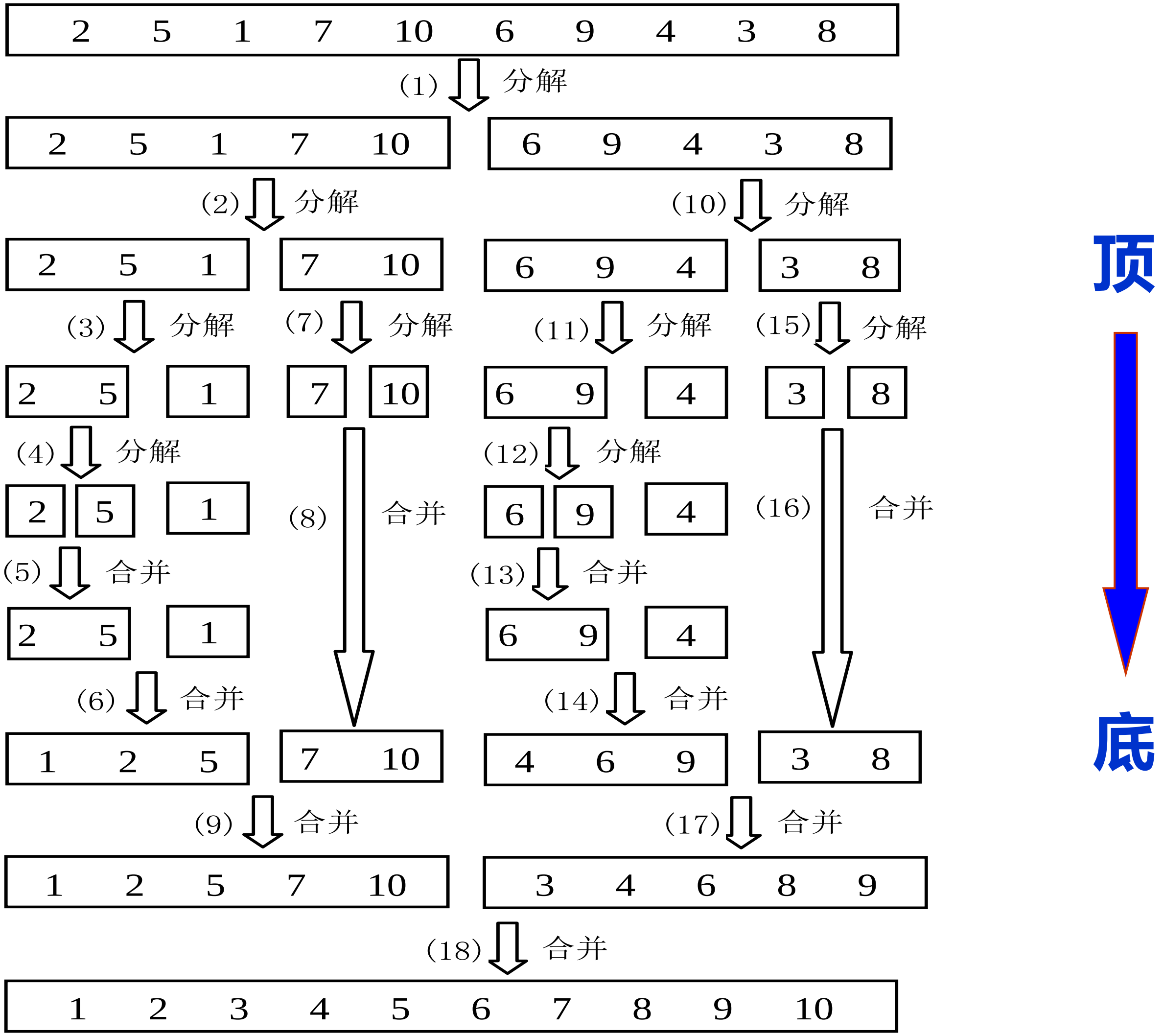
(6) 最后一步：将排好序的数据复制回原数组;

```
//将有序的a[low..mid]和a[mid+1..high]归并为有序的a[low..high]
void Merge(int a[], int low, int mid, int high)
{   int *tp;          // 临时存放合并后的数列
    int  i=low,       // 用来遍历左子集
    int  j=mid+1,     // 用来遍历右子集
    int  k=0;         // 用来遍历子集的合集

  //第一步：申请空间，使其大小为子集之和，用来存放合并后的序列
   tp=(int *)malloc((high-low+1)*sizeof(int));

  //第二步：遍历两个子集，选择相对小的元素放入到合并空间，并移动他的指针到下一位置
   while (i<=mid && j<=high)
       if (a[i]<=a[j])          //将第1子表中的元素放入tp中
       {  tp[k++]=a[i++];}
       else                     //将第2子表中的元素放入tp中
       {  tp[k++]=a[j++];}

  //倒数第二步：将另一序列剩下的所有元素直接复制到合并序列尾
   while (i<=mid)               //或者左子集有剩余：将第1子表余下部分复制到tp
   {     tp[k++]=a[i++];}
   while (j<=high)              //或者右子集有剩余：将第2子表余下部分复制到tp
   {     tp[k++]=a[j++];}

  //最后一步：将tmp复制回a中
   for (k=0,i=low;i<=high;k++,i++)
       a[i]=tp[k];
   free(tp);                    //释放tp所占内存空间
}
```

(a[i]<=a[j])?(tp[k++]=a[i++]):(tp[k++]=a[j++]);

1. 自顶向下的二路归并排序算法(采用递归)

例如，对于{2,5,1,7,10,6,9,4,3,8}序列，其排序过程如下图所示，图中圆括号内的数字指出操作顺序。

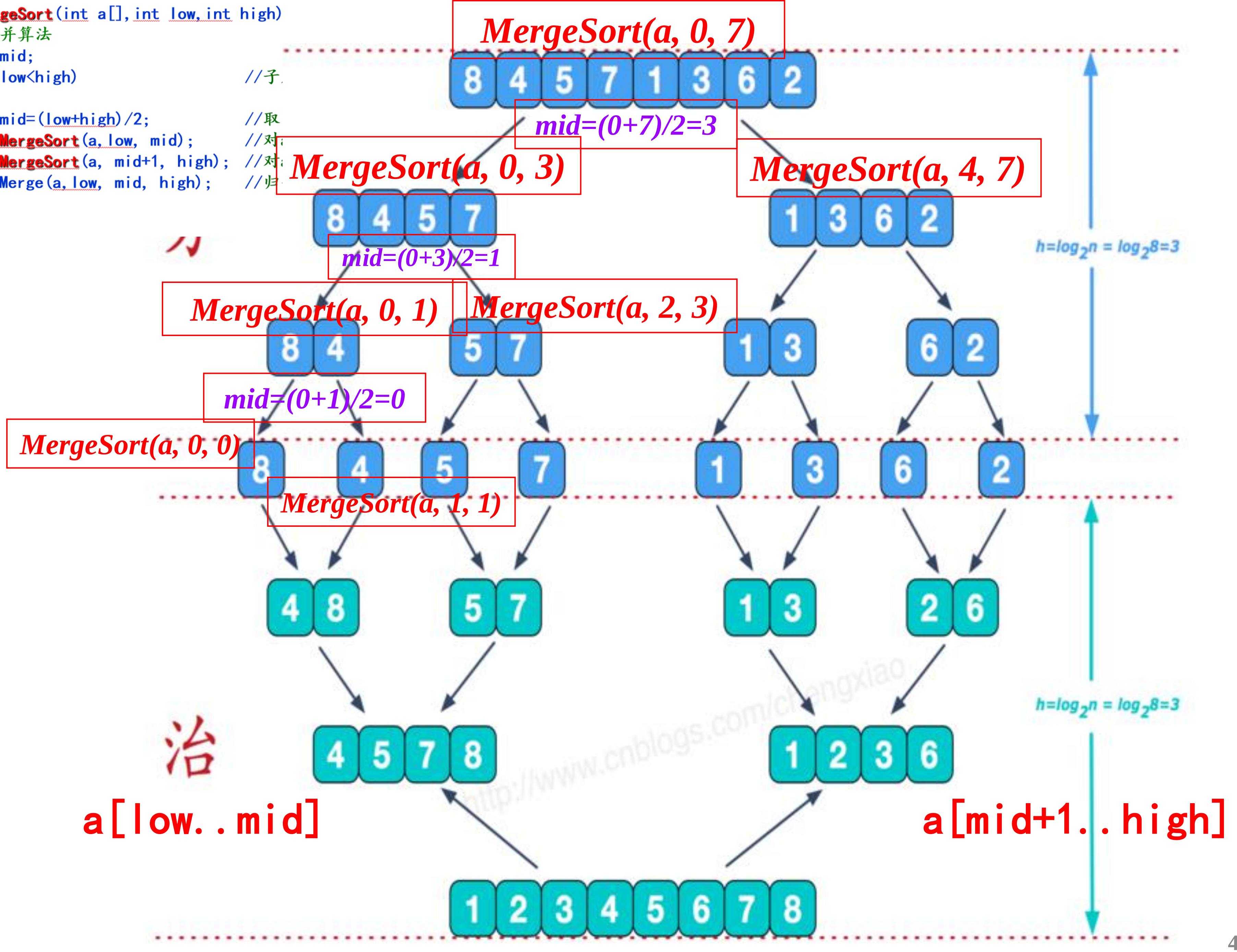


对应的二路归并排序算法如下：

```
//利用递归的自顶向下的归并算法
//  a：待排序数组（输入 && 输出）；
// low：数据区间低位下标（输入）；
// high：数据区间高位下标（输入）；
void MergeSort(int a[],int low,int high)
//二路归并算法
{   int mid;
    if (low<high)                  //子序列有两个或以上元素
    {
        mid=(low+high)/2;          //取中间位置
        MergeSort(a,low, mid);     //对a[low..mid]子序列排序
        MergeSort(a, mid+1, high); //对a[mid+1..high]子序列排序
        Merge(a,low, mid, high);   //归并
    }
}
```

问题：这个递归函数的递归基是什么？

```
void MergeSort(int a[],int low,int high)
//二路归并算法
{   int mid;
    if (low<high)                   //子
    {
        mid=(low+high)/2;           //取
        MergeSort(a,low, mid);      //对
        MergeSort(a, mid+1, high);  //对
        Merge(a,low, mid, high);    //归
    }
}
```



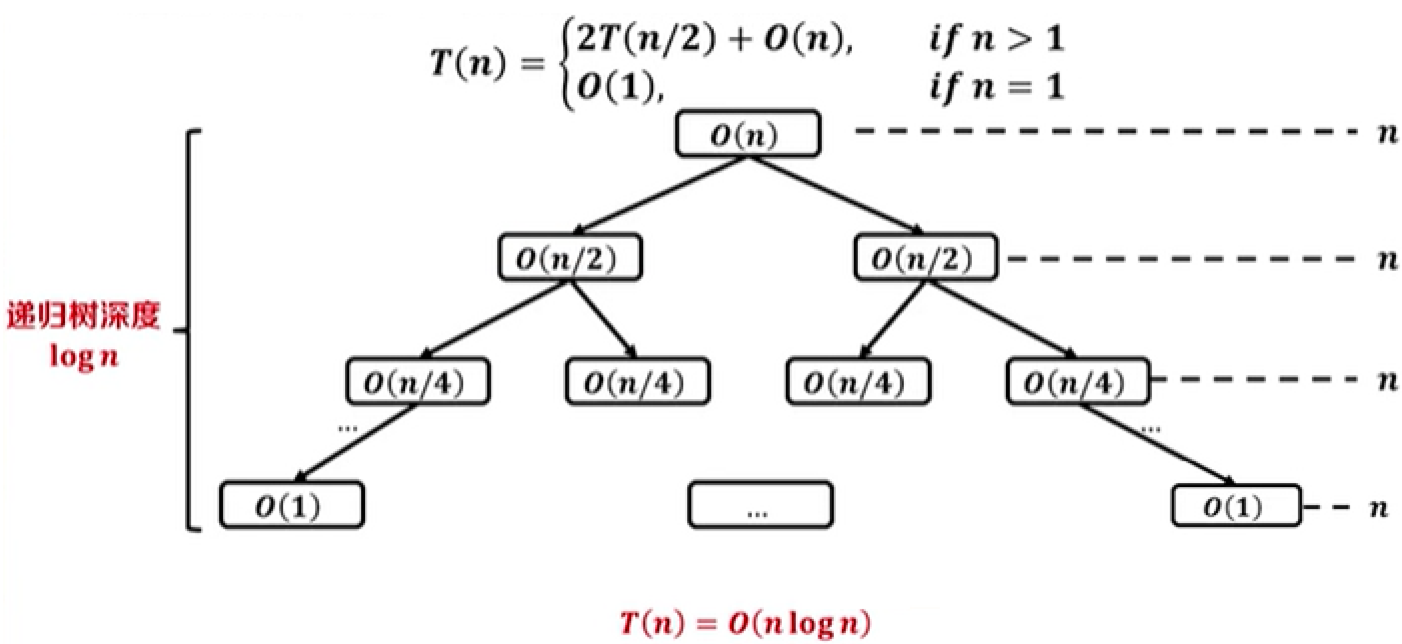
算法分析： 设MergeSort(a,0,n-1)算法的执行时间为$T(n)$，显然Merge(a,0,n/2,n-1)的执行时间为O(n)，所以得到以下递推式：

$T(n)=1$ 当$n=1$

$T(n)=2T(n/2)+O(n)$ 当$n>1$

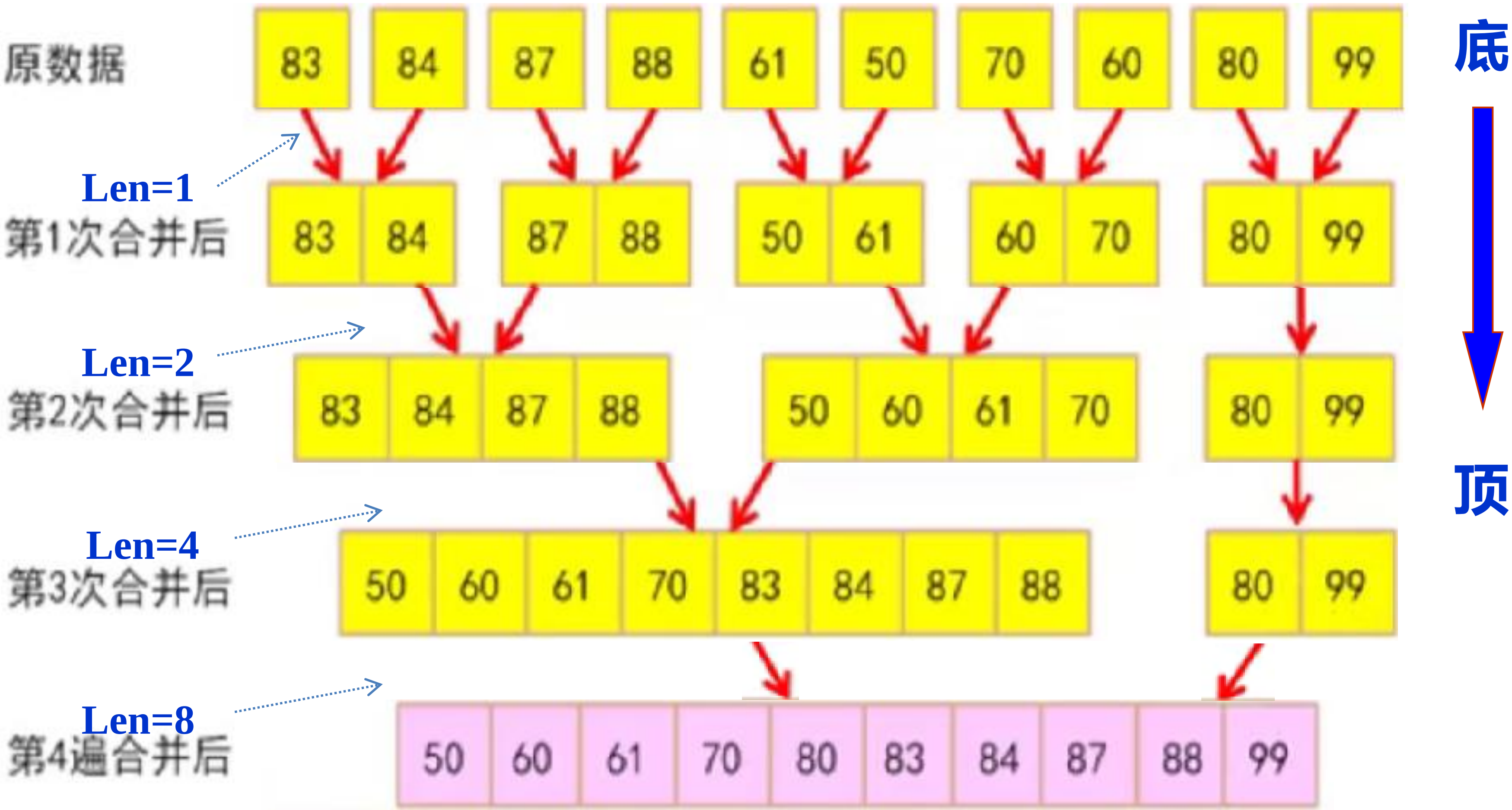
容易推出，$T(n)=O(n\log_2 n)$。

- 递归树法：用树的形式表示抽象递归



2. 自底向上的二路归并排序算法（采用迭代）

例如，对于{83,84,87,88,61,50,70,60,80,90}序列，其排序过程如图所示，图中方括号内是一个有序子序列。



二路归并排序的分治策略如下：

循环$\lceil \log_2 n \rceil$次，length依次取1、2、…、$\log_2 n$。每次执行以下步骤：

① 分解：将原序列分解成length长度的若干子序列。

② 求解子问题：将相邻的两个子序列调用Merge算法合并成一个有序子序列。

③ 合并：由于整个序列存放在数组中a中，排序过程是就地进行的，合并步骤不需要执行任何操作。

```
// MergeSort : 自底向上的归并算法,经过多趟归并进行排序
// a: 待排序数组（输入 && 输出）;
// n: 数组元素的个数;
void MergeSort(int a[],int n)
{
    int len; //归并len长度的两个子集合,len最大(high-low+1)/2;

    // 以len为长度，多次调用归并函数，进行归并
    for (len=1;len<n;len=2*len)
    {  // 先从步长为1开始归并，两两归并，归并为多组（2元素）子集
       // 之后归并步长为2进行归并，2+2，归并为多组（4元素）子集
       // 之后归并步长为4进行归并，4+4，归并为多组（8元素）子集
       // 多趟归并，总计需要⌈logn⌉趟
        MergePass(a, len, n);
     }
}
```

算法分析：对于上述二路归并排序算法，当有n个元素时，需要$\lceil \log_2 n \rceil$趟归并，每一趟归并，其元素比较次数不超过n-1，元素移动次数都是n，因此归并排序的时间复杂度为$O(n\log_2 n)$。

```
//一趟二路归并排序， 以2*len为跨度一个子集、一个子集地逐个归并
//a:  待排序数组，输入参数，同时也是输出参数。
//     a[0...len-1.len..2*len-1]、 a[2*len... 3*len-1.3*len...4*len-1]、 ...
//len: 归并长度，which is 1,2,4,8,...2*len<n
//n:   数组元素的个数;
void MergePass(int a[],int len, int n)
{   int i;

    //以下标0开始，逐个归并长度为len的两相邻子集，归并为2len长度的更大集合
    for (i=0; i+2*len-1<n; i=i+2*len)
    {
        //归并a[i.. i+len-1] 和 a[i+len.. i+2*len-1]
        Merge(a, i, i+len-1, i+2*len-1);
     }

    //剩余部分不够2个len长度，但如果超过了len长度(例如len=2，剩余了3个数),
    //可分为两个子集,前半部分长度为len，后半部分长度小于len,排序归并这两部分
    if (i+len-1 < n)
    {
       //归并a[i.. i+len-1] 和 a[i+len.. n-1]
        Merge(a, i, i+len-1, n-1);
     }
}
```

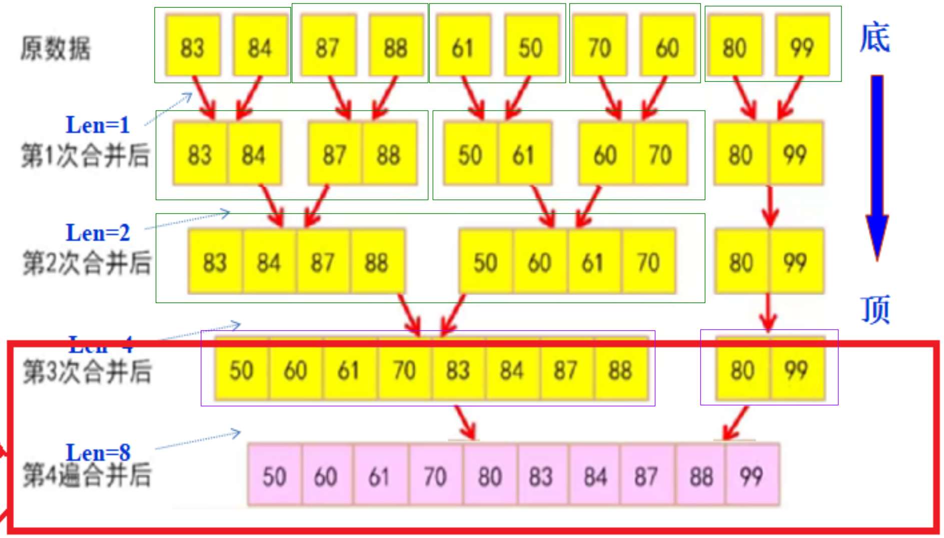
```
//一趟二路归并排序， 以2*len为跨度一个子集和一个子集地逐个归并
//a:  待排序数组，输入参数，同时也是输出参数。
//     a[0...len-1.len..2*len-1]、 a[2*len... 3*len-1.3*len...4*len-1]、 ...
//len: 归并长度，which is 1,2,4,8,...2*len<n
//n:   数组元素的个数；
void MergePass(int a[],int len, int n)
{   int i;

    //以下标0开始，逐个归并长度为len的两相邻子集，归并为2len长度的更大集合
    for (i=0; i+2*len-1<n;  i=i+2*len)
    {
        //归并a[i.. i+len-1] 和 a[i+len.. i+2*len-1]
        Merge(a, i, i+len-1, i+2*len-1);
    }

    //剩余部分不够2个len长度，但如果超过了len长度(例如len=2, 剩余了3个数),
    //可分为两个子集,前半部分长度为len, 后半部分长度小于len,排序归并这两部分
    if (i+len-1 < n)
    {
        //归并a[i.. i+len-1] 和 a[i+len.. n-1]
        Merge(a, i, i+len-1, n-1);
    }
}
```

45

减一的原因，都是因为下标是从0开始计数，例如6个数，下标应该是 0......5

2. 自底向上的二路归并排序算法

例如，对于{83,84,87,88,61,50,70,60,80,90}序列，其排序过程如图所示，图中方括号内是一个有序子序列。



最后这趟归并，len=8, 不够2个len(16个数)，所以上面的for循环不会被执行，
接着执行这个if语句， 这10个数 (8+2) ,虽然不够2个len(len=8)长度，但是超过了一个len(len=8)的长度，if条件成立，所以，执行 (i......i+len-1)区间的数（长度为len）和剩余的两个数（例子中的80、99）的归并。
这是一组满足if条件的例子，即：归并数列不够2个len（如果够2个len的话，就会执行上面的for循环，进行len 和 len 数列的归并），但是还够1个len + 剩余数（直到最后下标为n-1的数），则对他们进行归并！

常用的排序算法的时间复杂度和空间复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(nlog_2n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定	较复杂

归并排序需要临时空间存储数据

53

求解查找问题

2-3 二分搜索技术（折半查找）★

问题描述：

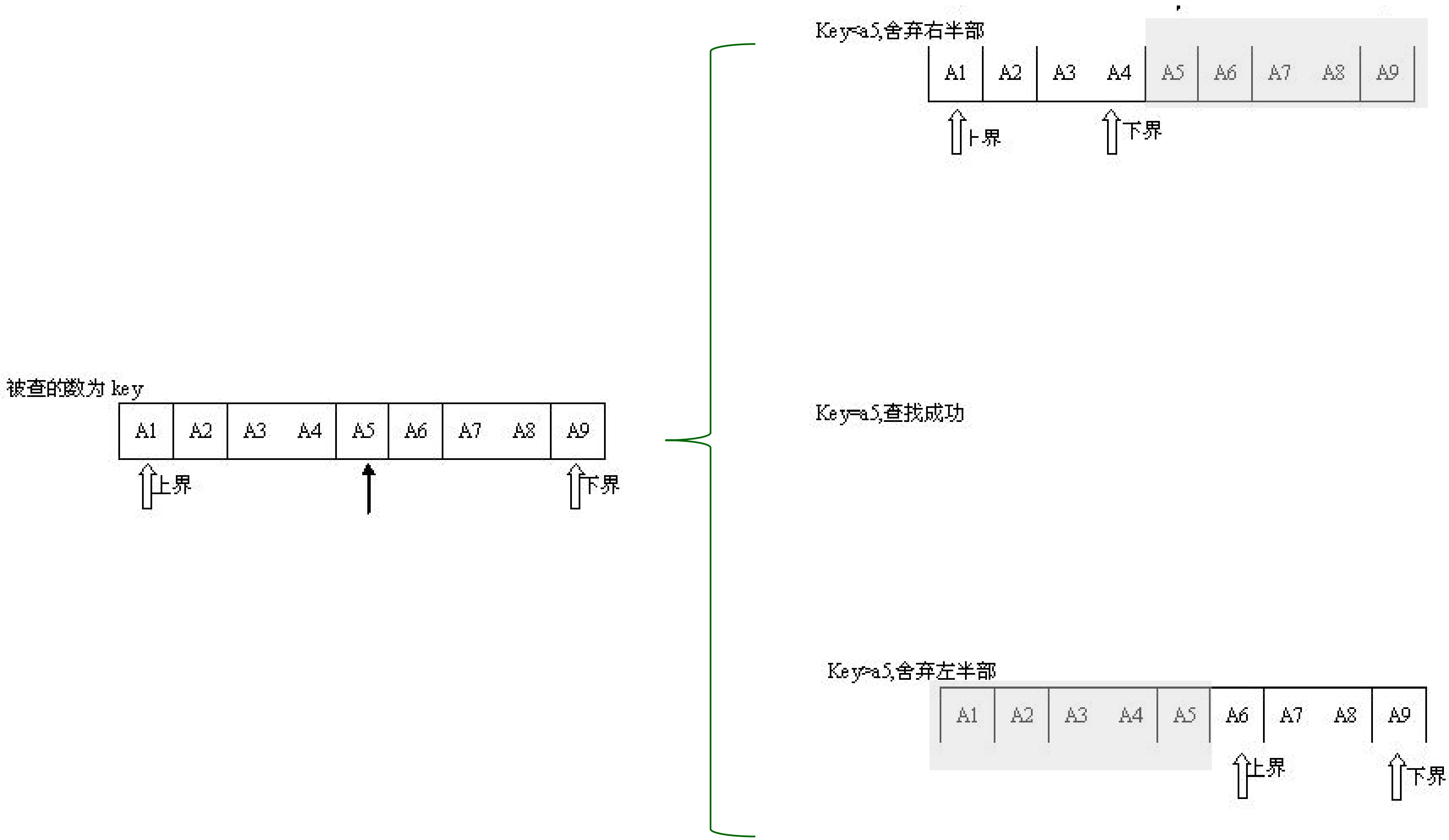
给定已按**升序排好序**的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。

问题分析：

- ✓ 该问题的规模缩小到一定的程度就可以容易地解决；
- ✓ 该问题可以分解为若干个规模较小的相同问题;
- ✓ 分解出的子问题的解可以合并为原问题的解；
- ✓ 分解出的各个子问题是相互独立的。

问题求解： 采用折半的二分查找方法。

折半查找过程的图解：



被查的数为 key
A1 A2 A3 A4 A5 A6 A7 A8 A9
上界
下界
Key<a5,舍弃右半部
A1 A2 A3 A4 A5 A6 A7 A8 A9
上界
下界
Key=a5,查找成功
Key>a5,舍弃左半部
A1 A2 A3 A4 A5 A6 A7 A8 A9
上界
下界

算法实现：

```
//拆半查找算法
//a: 输入参数，待查找数组
//low,high:输入参数，数组的上界（下标）和（下界）
//k:输入参数，待查找的元素
//返回：查找到元素的对应位置。未找到：返回-1
int BinSearch(int a[], int low, int high, int k)
{   int mid=-1;
    if (low<=high)              //当前区间存在元素时
    {   mid=(low+high)/2;       //求查找区间的中间位置
        if (a[mid]==k)          //找到后返回其下标mid
            return mid;

        if (a[mid]>k)          //当a[mid]>k时（可能位于左面，查左面）
            return BinSearch(a,low,mid-1,k);
        else                   //当a[mid]<k时（可能位于右面，查右面）
            return BinSearch(a,mid+1,high,k);
    }
    else return -1;   //若当前查找区间没有元素时返回-1
}
```

算法分析：折半查找算法的主要时间花费在元素比较上，对于含有*n*个元素的有序表，每执行一次算法的折半查找， 待搜索数组的大小减少一半。因此，在最坏情况下，循环被执行了O(logn) 次。循环体内运算需要O(1) 时间，因此整个算法在最坏情况下的计算时间复杂性为O(logn) 。

2-4 线性时间选择：寻找一个序列中第k小元素

问题描述：对于给定的含有n元素的无序序列，求这个序列中第*k*（1≤*k*≤*n*）小的元素。

例如，如下序列中查找第3小的元素：

{13、4、6、19、16、8、5、3、17、21}

求解方法1：

假设无序序列存放在*a*[0..*n*-1]中，若将*a*递增排序，则第*k*小的元素为*a*[*k*-1]。

求解方法2：随机划分线性选择，采用类似于快速排序的思想。

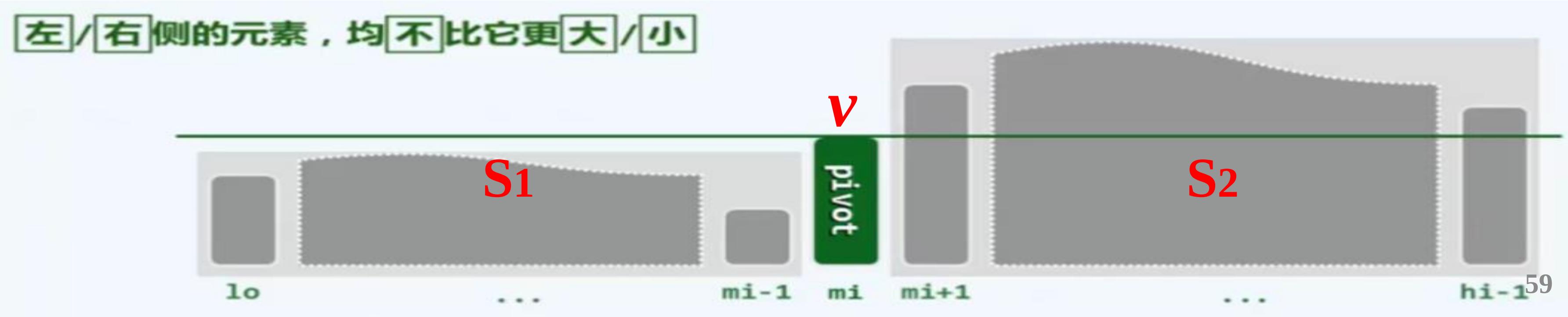
(1) 选取一个枢纽元 $v \in S$。

(2) 将集合 $S - \{v\}$ 分割成 S_1 和 S_2，就像我们在快速排序中所做的那样。

(3) 如果 $k \leq |S_1|$，那么第 k 个最小元必然在 S_1 中。

如果 $k = 1 + |S_1|$，那么枢纽元就是第 k 个最小元。

否则，这第 k 个最小元就在 S_2 中



算法实参考代码1：

注意确保K的有效性！

```
//在a[low..high]序列中找第k小的元素
int QuickSelect(int a[],int low,int high,int k)
{
    if (k-1>high || k-1<low)  // 确保K的有效性
    {
          return -1;
     }
    if (low < high)
    {
        int mid = partition(a, n, low, high);
        if (k-1 == mid ) //分割位就是要找的位置
        {
              return a[mid];
         }
        if (k-1<mid)    // 要找的元素位于在分割位的左侧
        {
           return QuickSelect(a, low, mid-1, k);
        }
        else            // 要找的元素位于在分割位的右侧
        {
           return QuickSelect(a, mid+1, high, k);
        }
    }
    else if ((low == high) && (low == k-1))  // 区间内只有一个元素，为a[k-1]（ low =high ）
    {    return a[low];
     }
    else
    {
          return -1;
     }
}
```

算法实现参考代码2:

```
//在a[left..right]序列中找第k小的元素
int QuickSelect(int a[],int left,int right,int k)
{
    int i = left, j = right, tmp = 0, mid = 0;
    if (left < right)
    {
        int pivot = a[lo];

        //---- 根据基准点进行划分, 定位中间位置i  ----
        while (i!=j)
        {
            while (i<j && a[j]>= pivot)
                j--;
            while (i<j && a[i]<= pivot)
                i++;

            swap(a,i,j);
        }

        //这时候, a[i]左面都比基准数小或相等, a[i]右面的都比基准位大或相等
        //----让原处于首位的基准 (轴) 位归位, 与a[i]互相交换 ----
        swap(a,lo,i);

        if (k-1 == mid ) //分割位就是要找的位置
        {
            return a[mid];
        }
        if ( k-1< mid)    // 要找的元素位子在分割位的左侧
        {
            return QuickSelect(a, left, mid-1, k);
        }
        else             // 要找的元素位子在分割位的右侧
        {
            return QuickSelect(a, mid+1, right, k);
        }
    }
    else if ((left == right) && (left == k-1))  // 区间内只有一个元素, 为a[k-1] ( low =high )
    {   return a[left];
    }
    else
    {
        return -1;
    }
}
```

求解方法2的举例：查找第3小的数字，x为随机选取的基准值

a=(3、 14、 11、 8、 17、 19、 5、 6、 9、 2)

长度为5

a=(3、8、5、6、2)

第6大数

x=(9)

a=(14、 11、 17、 19)

长度为3

a=(3、5、2)

第4大数

x=(6)

a=(8)

长度为2

a=(2、3)

第3大数

x=(5)

算法分析：对于QuickSelect(a,s,t,k)算法，设序列a中含有n个元素，其比较次数的递推式为：

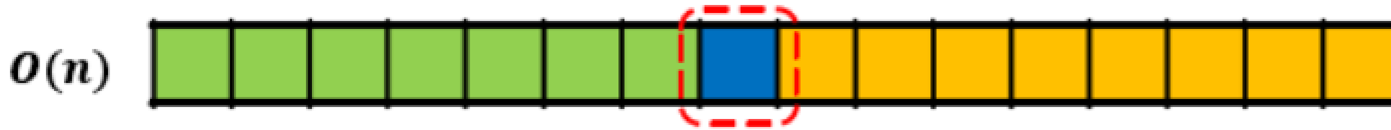
$$T(n)=T(n/2)+O(n)$$

可以推导出$T(n)=O(n)$，这是最好的情况，即每次划分的基准恰好是中位数，将一个序列划分为长度大致相等的两个子序列。

在最坏情况下，每次划分的基准恰好是序列中的最大值或最小值，则处理区间只比上一次减少1个元素，此时比较次数为$O(n^2)$。

在平均情况下该算法的时间复杂度为$O(n)$。

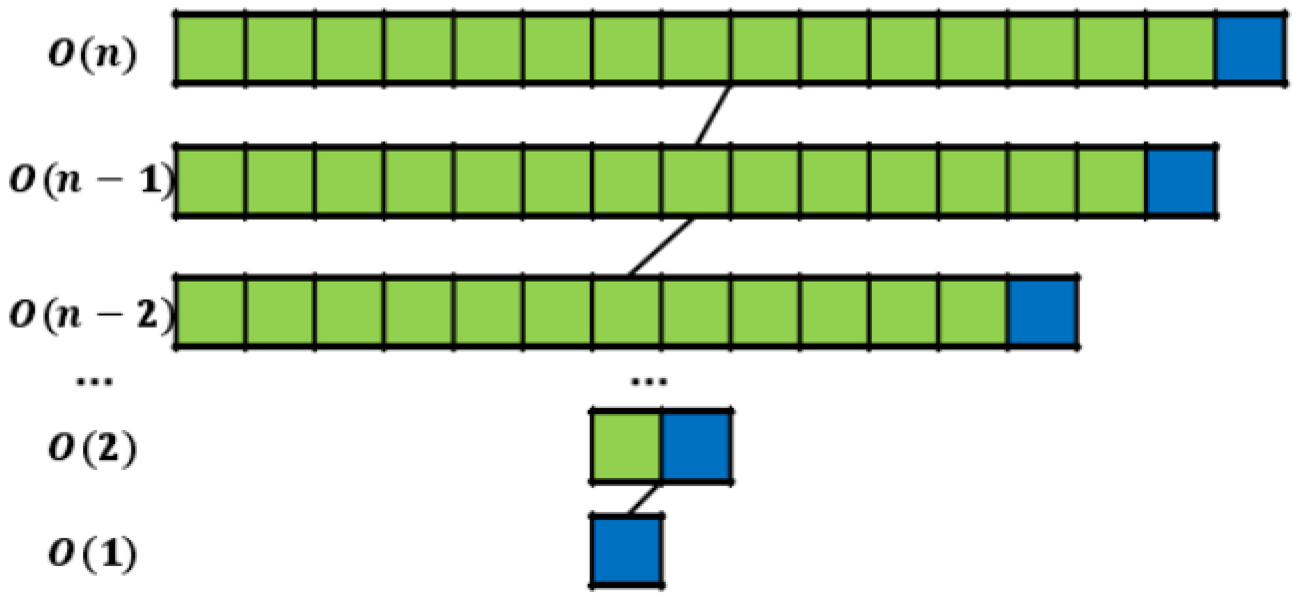
◆最好情况



第k小元素

- 时间复杂度：$T(n) = O(n)$

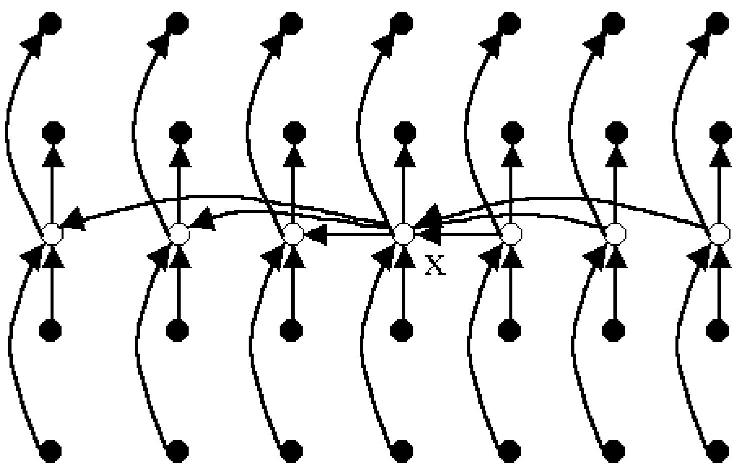
◆最坏情况



- 时间复杂度：$T(n) = \sum_{i=1}^{n} i \leq n^2 = O(n^2)$

问题求解2：改进算法，“5分法中位数的中位数”作为主元（枢纽元）利用中位数线性时间选择，***中位数***：是指将数据按大小顺序排列起来，形成一个数列，居于数列中间位置的那个数据。

具体思路：将n个输入元素划分成⌈n/5⌉个组，每组5个元素，只可能有一个组不是5个元素。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共⌈n/5⌉个。递归调用select来找出这⌈n/5⌉个元素的中位数。



参考：https://blog.csdn.net/liufeng_king/article/details/8480430

求解方法2的举例：查找第3小的数字

a=(3、 14、 11、 8、 17、 19、 5、 6、 9、 2)

a1=(3、 14、 11、 8、 17)

a2=(19、 5、 6、 9、 2)

中值元素集合=(11、6)

偶数的情况，取较大的11

长度为6

a1=(3、 8、 5、 6、 9、 2)

x=(11)

a2=(14、 17、 19、)

a11=(3、 8、 5、 6、 9)

a12=(2)

中值元素集合=(6、2)

偶数的情况，取较大的6

长度为3

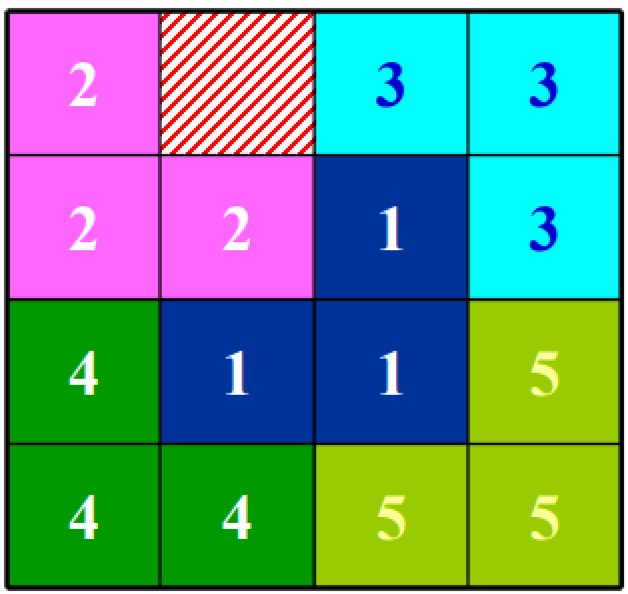
a11=(3、5、 2)

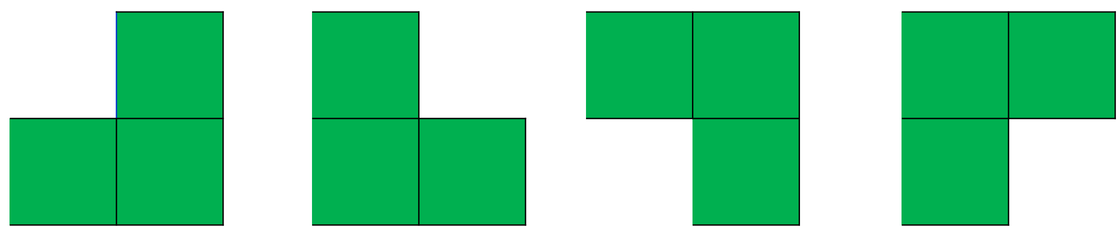
x=(6)

a12=(8、9)

2-5 棋盘覆盖

问题描述：在一个2^k * 2^k（k>0）个方格组成的棋盘中，有一个方格与其它的不同，若使用以下四种L型骨牌覆盖除这个特殊方格的其它方格，且任何2个L型骨牌不得重叠覆盖，如何覆盖。





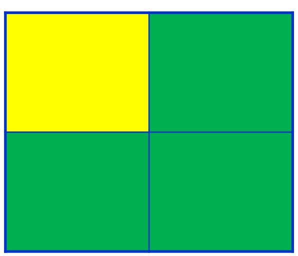
4中不同形态的L型骨牌

问题求解：实现的基本原理是将$2^k \times 2^k$的棋盘分成四块$2^{k-1} \times 2^{k-1}$的子棋盘，特殊方格一定在其中的一个子棋盘中。

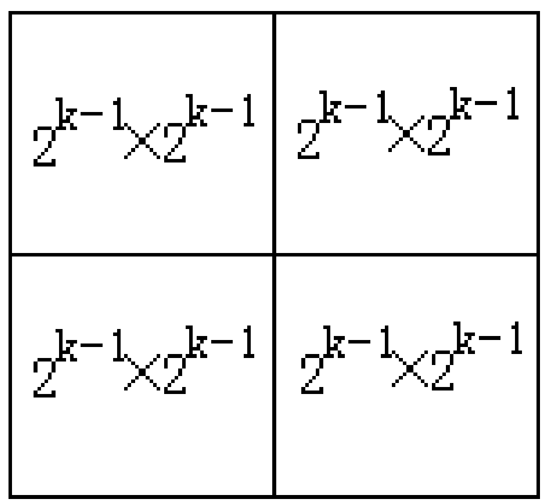
如果特殊方格在某一个子棋盘中：继续递归处理这个子棋盘，直到这个子棋盘中只有一个方格为止。

如果特殊方格不在某一个子棋盘中：将这个子棋盘中的**汇合位置设为骨牌号**，将这个无特殊方格的子棋盘转换为有特殊方格的子棋盘，原问题转换为4个小一级规模的相同问题。然后再递归处理这个子棋盘。

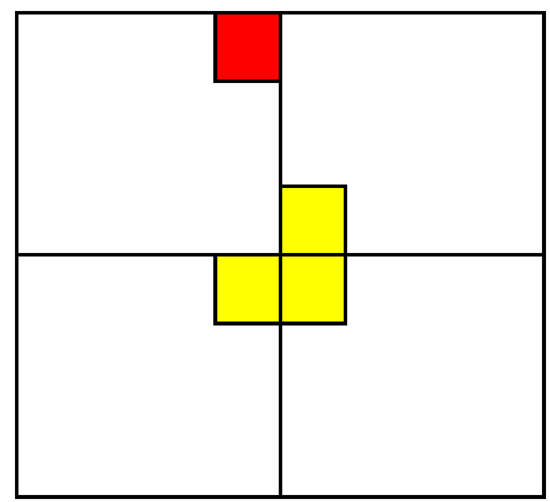
以上原理如下图所示：



最简情况

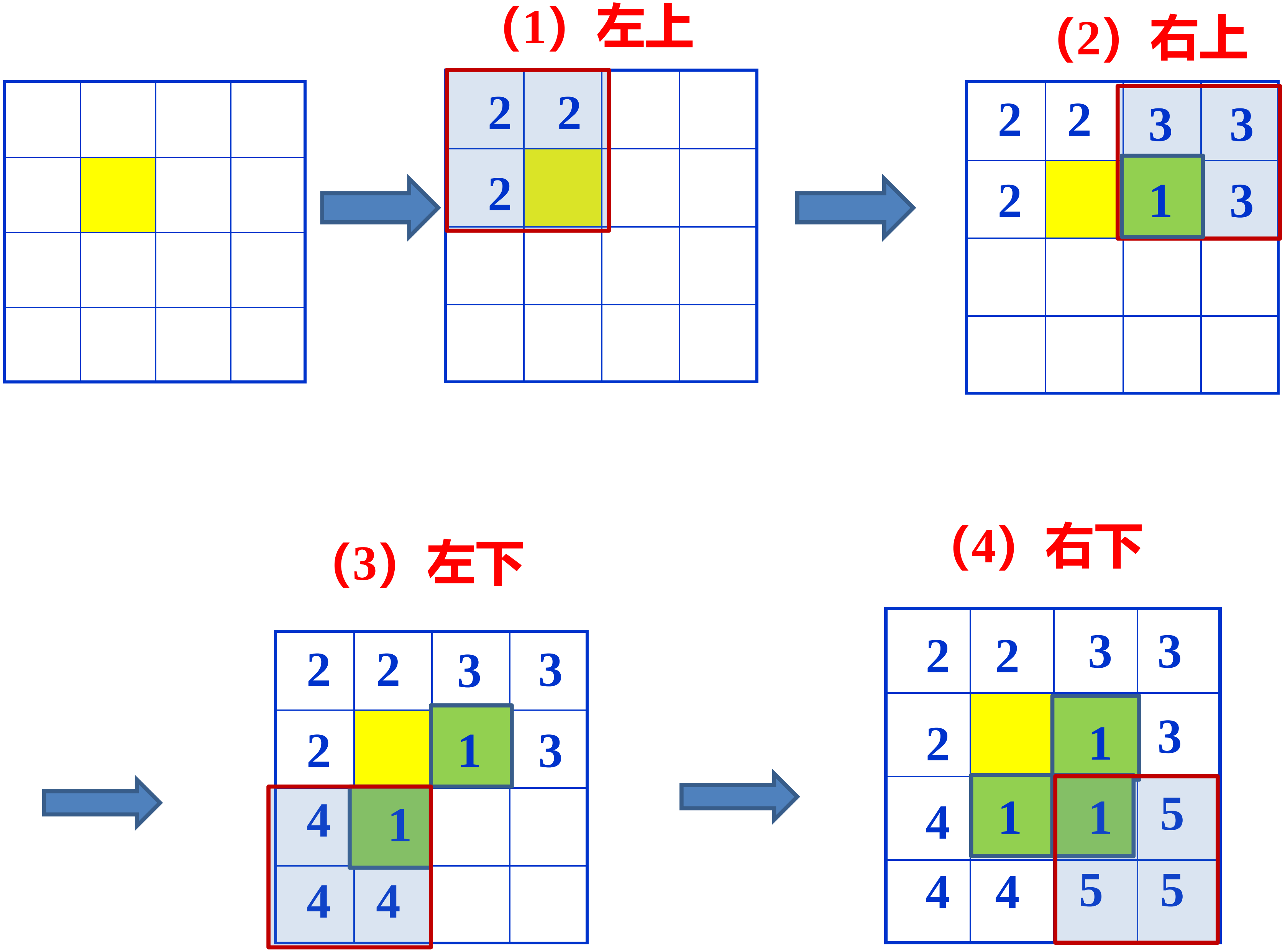


(a)



(b)

L形避开特殊方格


(1) 左上
2 2
2
(2) 右上
2 2 3 3
2 1 3
(3) 左下
2 2 3 3
2 1 3
4 1
4 4
(4) 右下
2 2 3 3
2 1 3
4 1 1 5
4 4 5 5

函数输入：tr，tc，dr，dc，size含义如下：

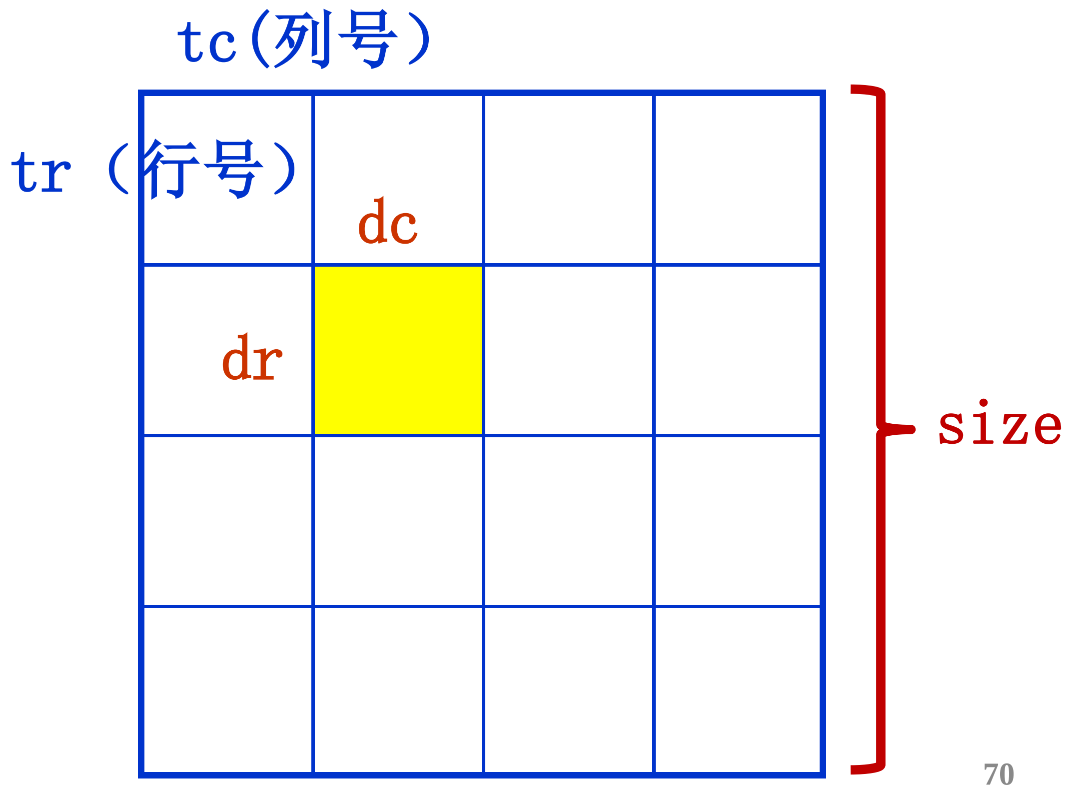
tr: 棋盘左上角方格的行号（下标从0开始）

tc: 棋盘左上角方格的列号（下标从0开始）

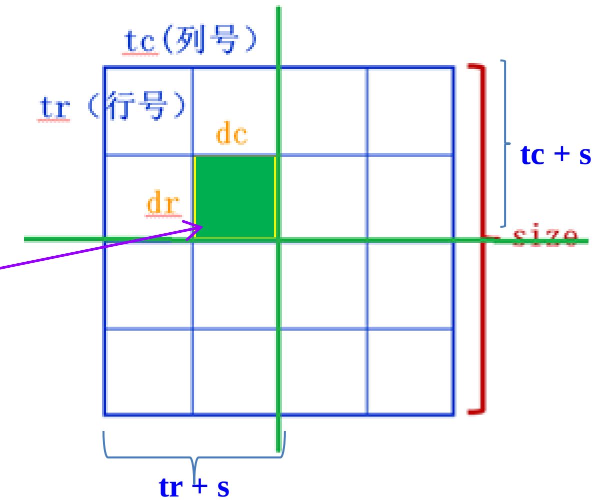
dr: 特殊方格所在的行号

dc: 特殊方格所在的列号

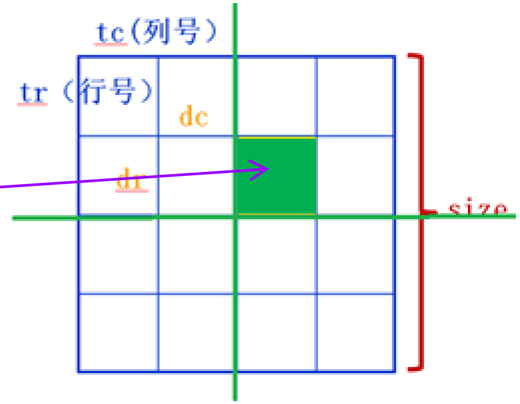
size: 方形棋盘的边长



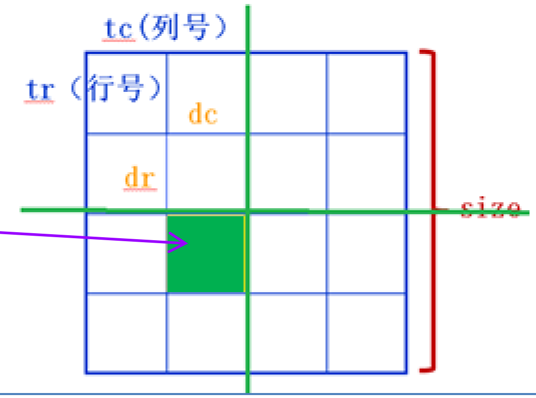
```
// tr, tc:  棋盘左上角的行号、列号（下标从0开始）
// dr, dc  特殊方格所在的行号、列号（下标从0开始）
// size:    方形棋盘的边长， 必须是2, 4, 8, 16...
void chessBoard(int tr, int tc, int dr, int dc, int size)
  {
     if (size == 1)   return; // 递归基
     int t = tile++;             // L型骨牌号, tile是全局变量，初始值为0，此处开始编号1
      s = size/2;                // 分割棋盘后的size

    // 覆盖左上方子棋盘
    if (dr < tr + s && dc < tc + s)    // 特殊方格在此棋盘中
          chessBoard(tr, tc, dr, dc, s);
    else {                                    // 此棋盘中无特殊方格
      // 用 t 号L型骨牌覆盖右下角
      board[tr + s - 1][tc + s - 1] = t;
      // 覆盖其余方格
      chessBoard(tr, tc, tr+s-1, tc+s-1, s);   // tr+s-1, tc+s-1成为新的特殊方格
    }
```



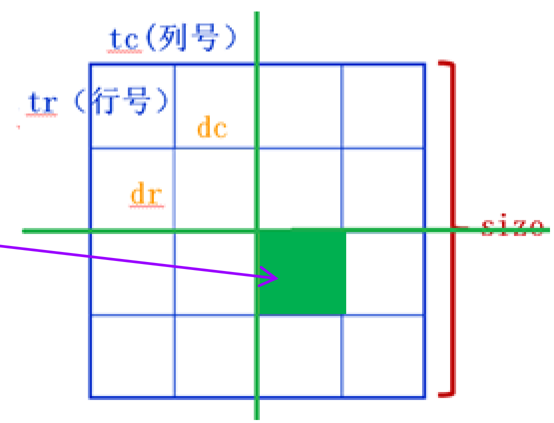
```
// 覆盖右上方子棋盘
if (dr < tr + s && dc >= tc + s)    // 特殊方格在此棋盘中
   chessBoard(tr, tc+s, dr, dc, s);
else {                               // 此棋盘中无特殊方格
   // 用 t 号L型骨牌覆盖左下角
   board[tr + s - 1][tc + s] = t;
  // 覆盖其余方格
   chessBoard(tr, tc+s, tr+s-1, tc+s, s);
 }
```




```
 // 覆盖左下方子棋盘
if (dr >= tr + s && dc < tc + s)   // 特殊方格在此棋盘中
   chessBoard(tr+s, tc, dr, dc, s);
else {                              // 此棋盘中无特殊方格
  // 用 t 号L型骨牌覆盖右上角
   board[tr + s][tc + s - 1] = t;
   // 覆盖其余方格
   chessBoard(tr+s, tc, tr+s, tc+s-1, s);
 }
```




```
// 覆盖右下方子棋盘
if (dr >= tr + s && dc >= tc + s) // 特殊方格在此棋盘中
    chessBoard(tr+s, tc+s, dr, dc, s);
else {                            // 此棋盘中无特殊方格
   // 用 t 号L型骨牌覆盖左上角
   board[tr + s][tc + s] = t;
   // 覆盖其余方格
   chessBoard(tr+s, tc+s, tr+s, tc+s, s);}
 }

} // chessBoard函数结束
```



2-6 循环赛日程表

问题描述：

对于 $n=2^k$ 个选手，设计一个满足以下要求的比赛日程表：
(1)每个选手必须与其他n-1个选手各赛一次；
(2)每个选手一天只能赛一次；
(3)循环赛一共进行n-1天。

问题求解：

分治思想：
(1)将所有的选手分为两半，n个选手的比赛日程表就可以通过为n/2个选手设计的比赛日程表来决定；
(2)递归地用对选手进行分割，直到只剩下2个选手时；

$\boxed{n=2} \longrightarrow$

选手	第一天赛事
1	2
2	1

最简情况

$n = 2^2 = 4$人的情况

	辅助	1天	2天	3天
1号	1	2	3	4
2号	2	1	4	3
3号	3	4	1	2
4号	4	3	2	1

$n = 2^3 = 8$人的情况

左上角与左下角的两小块分别为选手1至选手4和选手5至选手8前3天的比赛日程。据此，将左上角小块中的所有数字按其相对位置抄到右下角，又将左下角小块中的所有数字按其相对位置抄到右上角，这样我们就分别安排好了选手1至选手4和选手5至选手8在后4天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

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

$n = 2^3 = 8$人的情况(下图为自上而下的拷贝)

	辅助	1天	2天	3天	4天	5天	6天	7天
m=1 1号	1	2	3	4	5	6	7	8
2号	2	1	4	3	6	5	8	7
m=2 3号	3	4	1	2	7	8	5	6
4号	4	3	2	1	8	7	6	5
m=4 5号	5	6	7	8	1	2	3	4
6号	6	5	8	7	2	1	4	3
7号	7	8	5	6	3	4	1	2
8号	8	7	6	5	4	3	2	1

算法实现（为了便于理解，a数组的第0行舍弃不用）：

```
// n = 2^k 个人的循环赛日程表算法
// a[i][j]的值，表示第i个选手在第j天的比赛对手
void SetTable(int k, int n, int a[][])
{
   for(int j=1; j<=n; j++)  //首先设置日程表第一行
      a[1][j]=i;

   int m = 1 //m记录了填充的起始行(m+1)，同时也是填充块数的宽度，从最初行（宽度）1开始填充，
   for(int s=1; s<=k; s++)  //这种填充，从上到下，共完成k次(指数)大循环
   {
      n /= 2;                        //n: n/2、n/4、n/8…，记录填充块的数目，不断在减半
      for(int t=1; t<=n; t++)        //t从左到右n次循环，每个大循环里，有n个小的填充块（长度2m）
      {
         for(int i=m+1; i<=2*m; i++)     //i: 控制行m=1(2:2),m=2(3:4),m=4(5:8),m=8(9:16)…
         {                               //     i每取一（行）值，j负责从m+1扫到2m（一个填充块）
            for(int j=m+1; j<=2*m; j++) //j: 控制列，这个循环扫过长度为m的填充块！整长为2m!
            {
               a[i][j+(t-1)*m*2]    = a[i-m][j+(t-1)*m*2-m]; //右下角 <= 左上角值（上偏m位，左偏m位）
               a[i][j+(t-1)*m*2-m] = a[i-m][j+(t-1)*m*2];   //左下角 <= 右上角值（上偏m位）
            }
         }
      }
      m *= 2; //m=1、2、4、8、…
   }
}
```

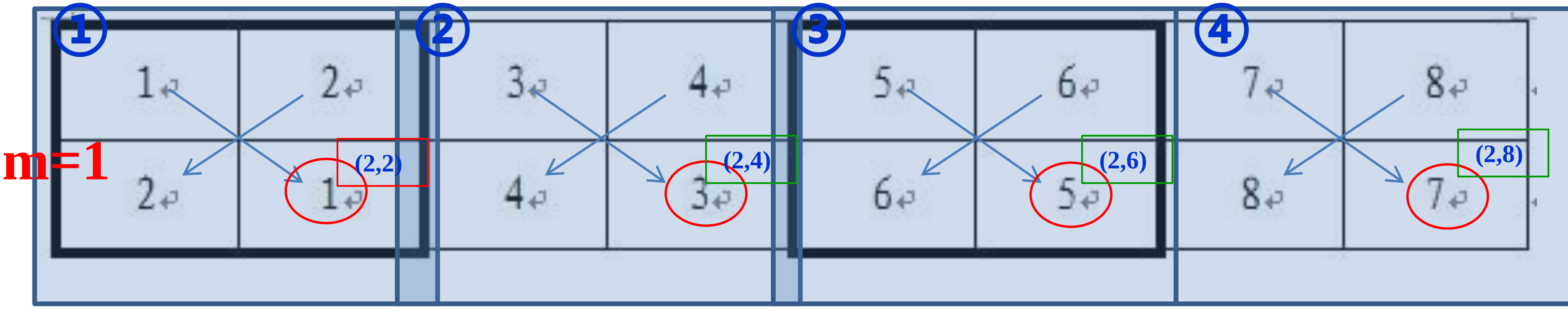
负责右半部分拷贝到左半部分

负责左半部分拷贝到右半部分

自左到右随着t的增加，列位置偏移(t-1)*2m

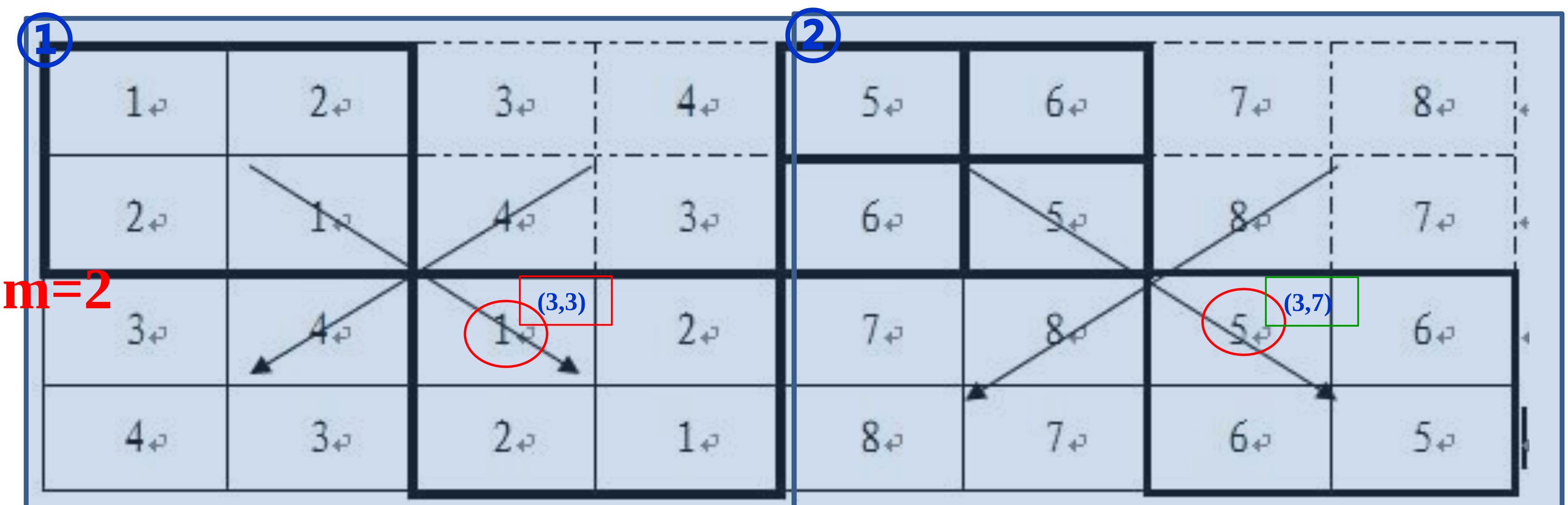
(1) s=1，根据初始化的第一行，填充第二行

m=1(行)，（n=4块、t循环4次)，i循环(2行:2行)，j循环同i



(2) s=2，进行第二部分填充，根据以上第1~2行值填充第3~4行

m=2(行)，（ n=2块、t循环2次)，i循环(3行:4行)，j循环同i



```
// n = 2^k 个人的循环赛日程表算法
// a[i][j]的值，表示第i个选手在第j天的比赛对手
void SetTable(int k, int n, int a[][])
{
  for(int j=1; j<=n; j++)  //首先设置日程表第一行
    a[1][j]=i;

  int m = 1 //m记录了填充的起始行(m+1)，同时也是填充块数的宽度，从最初行（宽度）1开始填充，
  for(int s=1; s<=k;  s++)  //这种填充，从上到下，共完成k次(指数)大循环
  {
    n /= 2;                  //n: n/2、n/4、n/8…，记录填充块的数目,不断在减半
    for(int t=1; t<=n; t++)  //t从左到右n次循环，每个大循环里，有n个小的填充块（长度2m）
    {
      for(int i=m+1; i<=2*m; i++)   //i：控制行m=1(2:2),m=2(3:4),m=4(5:8),m=8(9:16) …
      {                             //   i每取一（行）值，j负责从m+1扫到2m（一个填充块）
        for(int j=m+1; j<=2*m; j++) //j：控制列，这个循环扫过长度为m的填充块！整长为2m！
        {
          a[i][j+(t-1)*m*2]    = a[i-m][j+(t-1)*m*2-m];  //右下角 <= 左上角值（上偏m位，左偏m位）
          a[i][j+(t-1)*m*2-m] = a[i-m][j+(t-1)*m*2];    //左下角 <= 右上角值（上偏m位）
        }
      }
    }
    m *= 2; //m=1、2、4、8、…
  }
}
```

负责右半部分拷贝到左半部分

负责左半部分拷贝到右半部分

自左到右随着t的增加，列位置偏移(t-1)*2m

i: 一个复制块的行循环

j: 一个复制块的列循环

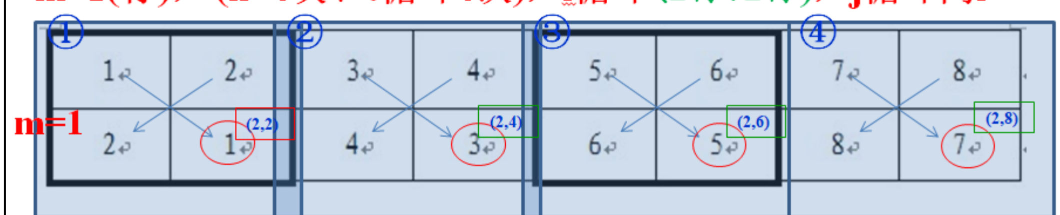
s:自上而下复制动作做几次

t: 自左到右复制动作做几次

i，j: 控制一个复制块

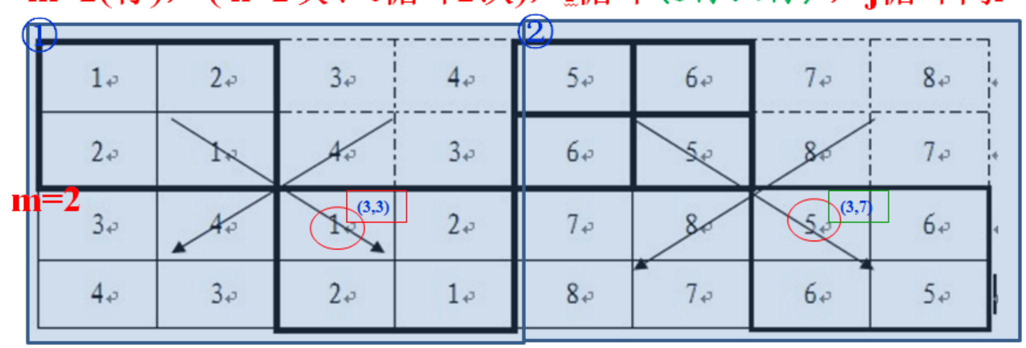
(1) s=1，根据初始化的第一行，填充第二行

m=1(行)，（n=4块、t循环4次)，i循环(2行:2行)，j循环同i



(2) s=2，进行第二部分填充，根据以上第1~2行值填充第3~4行

m=2(行)，（n=2块、t循环2次)，i循环(3行:4行)，j循环同i



(3) s=3，最后是第三部分，填充第5~8行：

m=4(行)， n=1块、t=1：自左到右循环拷贝1次，i循环（5行:8行），j循环同i

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

m=4

(5,5)

i循环
m+1
…
2m

a[i][j+(t-1)*m*2] = a[i-m][j+(t-1)*m*2-m];

a[i][j+(t-1)*m*2-m] = a[i-m][j+(t-1)*m*2];

总结

- 递归的概念
- 分治算法的设计思想
- 分治算法的做题思路
  - 快速排序
  - 归并排序
  - 二分查找特定元素
  - 查找第K小的数
  - 循环赛日程表

第一回 作业

- 完成以下算法代码
  - 快速排序
  - 归并排序
  - 折半查找
  - 线性时间选择
- 熟悉以上对应的算法（模拟演练）；
- 测试并观察特殊情况下（序列已经排好序、序列全部为相同数据等）的程序执行情况；
- 比较以上几种排序方法的时间效率；
- 2~3人一组，完成实验报告，报告内容：使用线性时间选择算法查找第k小数 与 先排序再获得第k小数 的效率比较。
- 格式不限，但要说明问题；
- 提交方式，电子文档，注意文档size不要超过1M；发送至网易邮箱：sftest_xgd@163.com，以群公告为准。
- 注意：编程时一定要选取便于跟踪调试代码的开发环境。
- 思考问题：第一节课提出的分卷子排序问题，如何分效率高？

赠送思考题

题目1：

一个有n级的台阶，一次可以走1级、或2级，问走完n级台阶有多少种走法?

题目2：

有没有看过具有递归情节的影视作品?

恐怖邮轮

END.