分治法

无论人们在祈祷什么，他们总是在祈祷一个奇迹，每一个祈祷都可以简化为：伟大的上帝啊，请让两个二相加不等于四吧。

——屠格涅夫

"二加二得四，

四加四得八，

八加八得十六，

依次重复下去！"

——J.普雷韦《写作专栏》

思考

- 假如你正在为一投资公司咨询。他们正在做模拟，对一给定的股票连续观察n天，记为i=1,2,…,n;对每天i,该股票每股的价格p(i)。假设在这个时间区间内，在某一天他们想买1000（第i天）股而在另一天（第j天）卖出所有这些股。为得到最多收益，他们应什么时候买什么时候卖？请设计算法在O(nlogn)时间内找到正确的i与j.

引子

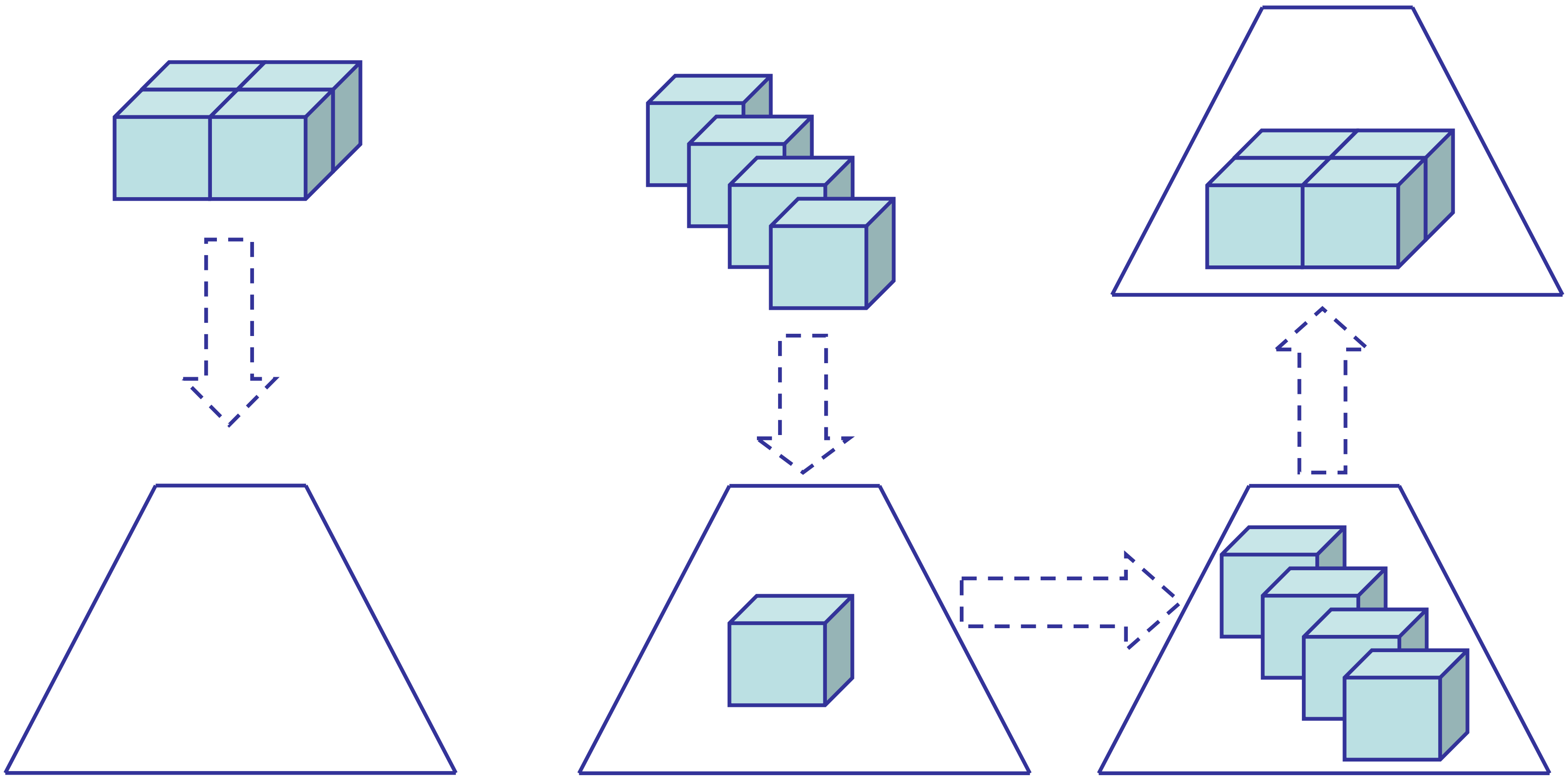
当面临一个复杂问题时，该如何下手呢？

把问题简化，即将复杂的大问题分解成简单的小问题，然后分而治之。

- 治国
  - ✓ 孙子兵法“凡治众如治寡，分数是也 ”
  - ✓ 国家→省→县→乡→村

- **齐家**

  ✓ *乔迁之喜，搬迁一个大组合柜*



- **算法设计领域**
  - ✓ 10000000万个整数的排序
  - ✓ 一个地图中10000000万个点，找出最近两个点

一个简单例子：**金块问题**

- 有一个老板有一袋金块。每个月将有两名雇员会因其优异的表现分别被奖励一个金块。按规矩，排名第一的雇员将得到袋中最重的金块，排名第二的雇员将得到袋中最轻的金块。
- 如果有新的金块周期性的加入袋中，则每个月都必须找出最轻和最重的金块。
- 假设有一台比较重量的仪器，我们希望用最少的比较次数找出最轻和最重的金块。

方法1

- 假设袋中有***n*** 个金块。可以通过***n*-1**次比较找到最重的金块。
- 找到最重的金块后，可以从余下的***n*-1**个金块中用类似的方法通过***n*-2**次比较找出最轻的金块。这样，比较的总次数为**2*n*-3**。

方法1

- 假设袋中有***n*** 个金块。可以通过***n*-1**次比较找到最重的金块。
- 找到最重的金块后，可以从余下的***n*-1**个金块中用类似的方法通过***n*-2**次比较找出最轻的金块。这样，比较的总次数为**2*n*-3**。

找金块的示例图

MAX=10

\- - - - - -

8 2 4 5 3 9 1

比较7次后得出MAX=10

MIN=8

\- - - - - -

2 4 5 3 9 1

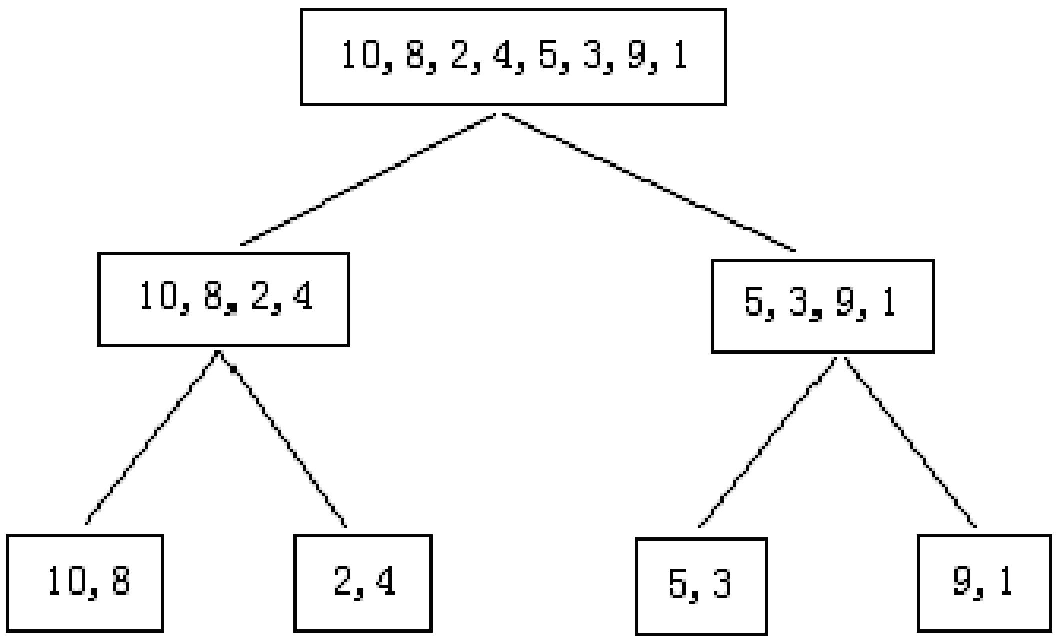
比较6次后得出MIN=1

共比较13次

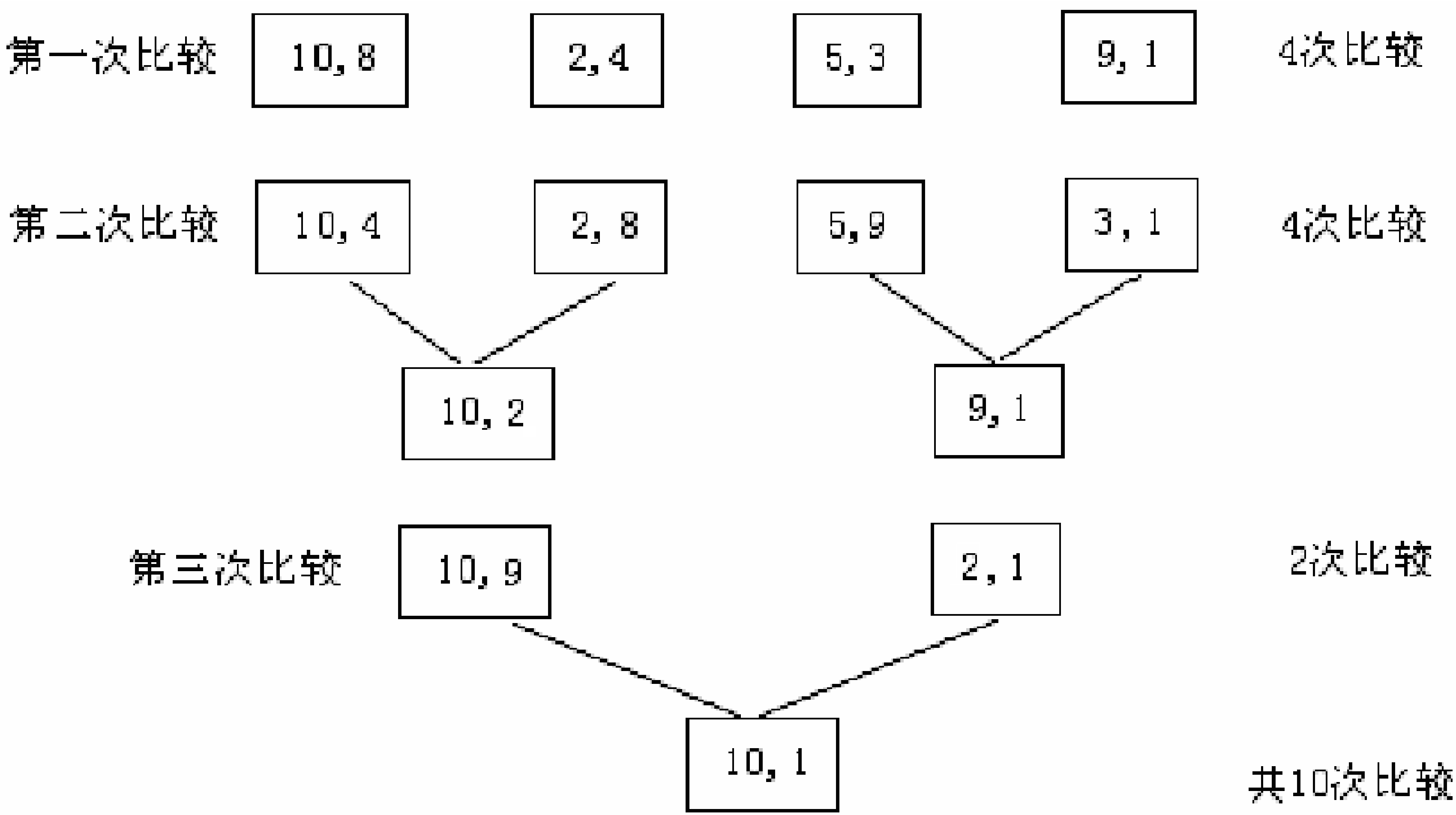
方法2：

- ***n*≤2**，识别出最重和最轻的金块，一次比较就足够了。
- ***n*＞2**，
  - 第一步，把这袋金块平分成两个小袋***A***和***B***。
  - 第二步，分别找出在***A***和***B***中最重和最轻的金块。设***A***中最重和最轻的金块分别为***HA*** 与***LA***，以此类推，***B***中最重和最轻的金块分别为***HB*** 和***LB***。
  - 第三步，通过比较***HA*** 和***HB***，可以找到所有金块中最重的；通过比较***LA*** 和***LB***，可以找到所有金块中最轻的。在第二步中，若***n*＞2**，则递归地应用分而治之方法。

分治过程



比较过程



第一次比较
10，8
2，4
5，3
9，1
4次比较
第二次比较
10，4
2，8
5，9
3，1
4次比较
10，2
9，1
第三次比较
10，9
2，1
2次比较
10，1
共10次比较

分析

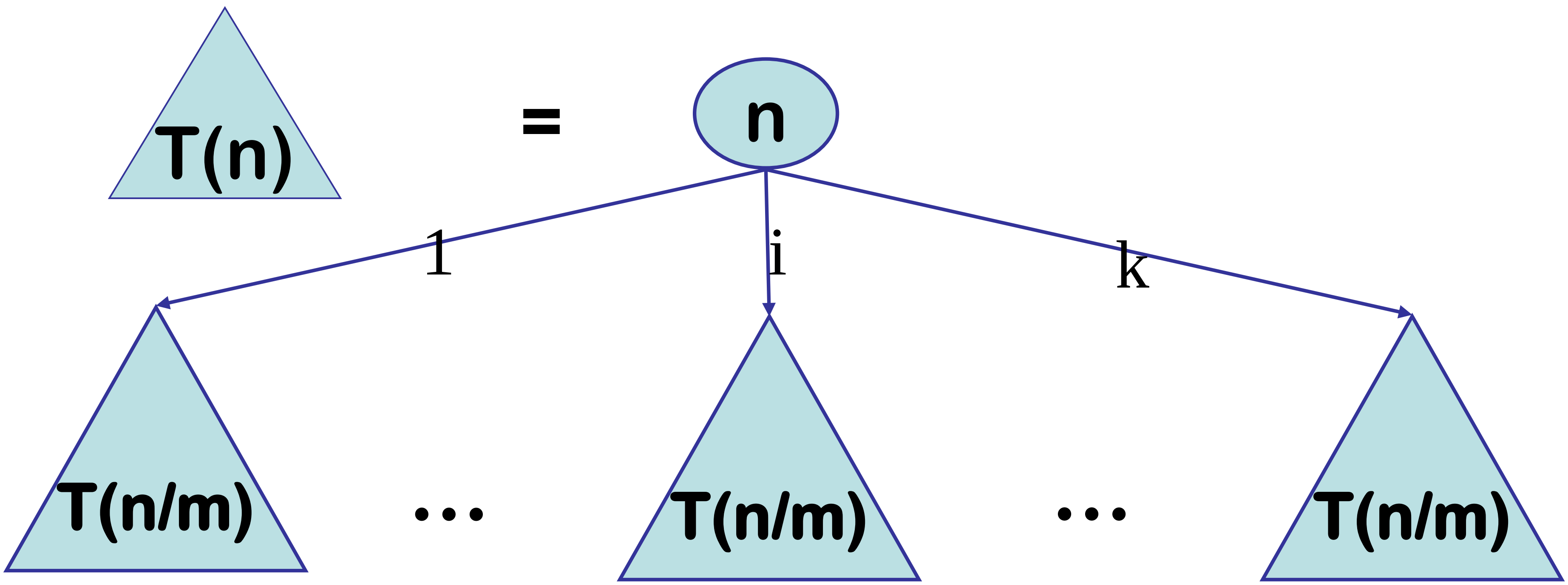
- **当有8个金块的时候，方法1需要比较13~14次，方法2只需要比较10次,那么形成比较次数差异的根本原因在哪里?**

- **其原因在于方法2对金块实行了分组比较。**

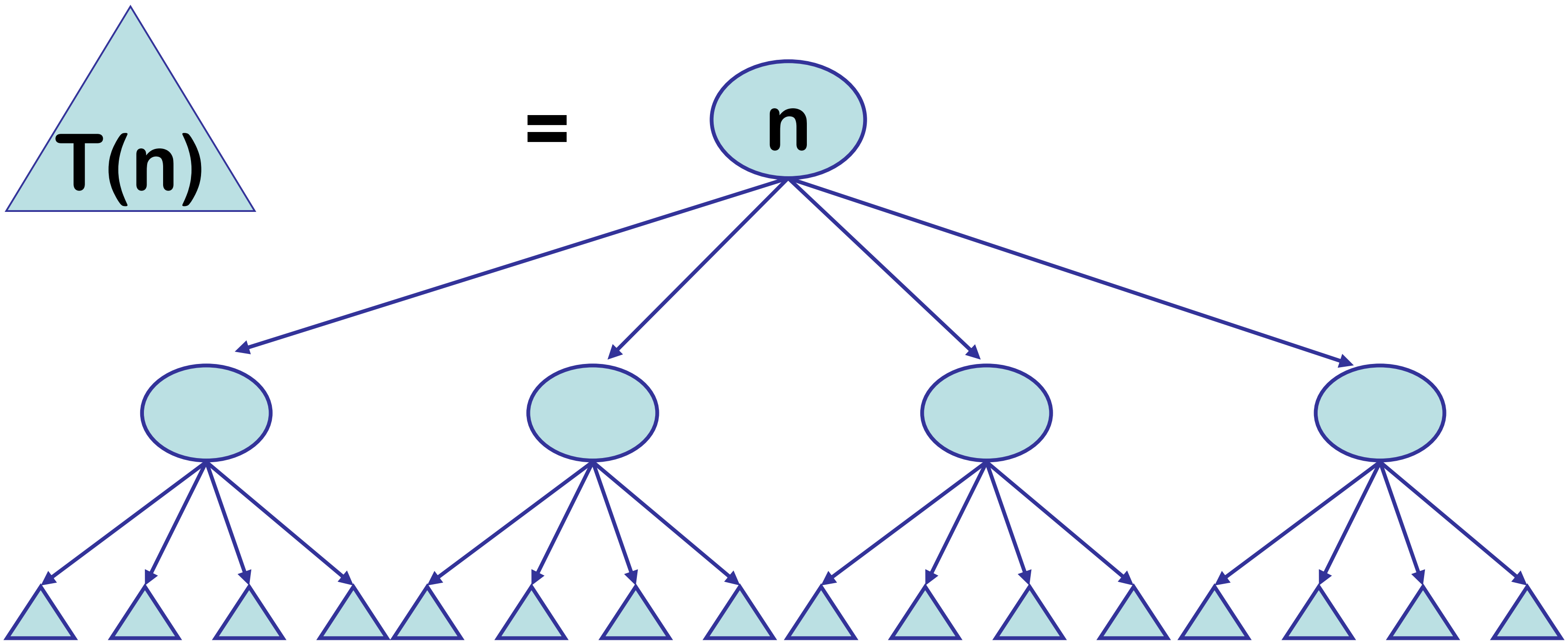
- **在本例中，使用方法2比方法1少用了2 5％左右的比较次数。**

分治法算法思想

- 分—将要求解的较大规模的问题分割成k个更小规模的子问题。



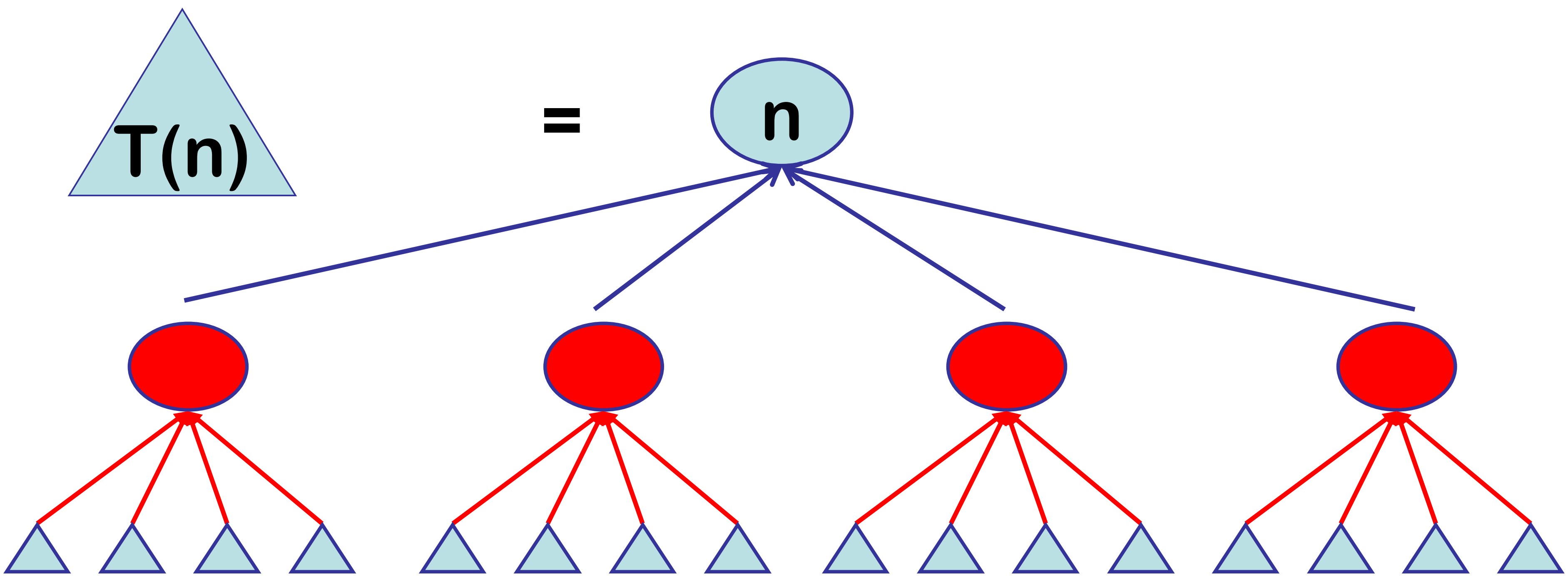
- **治**—对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。



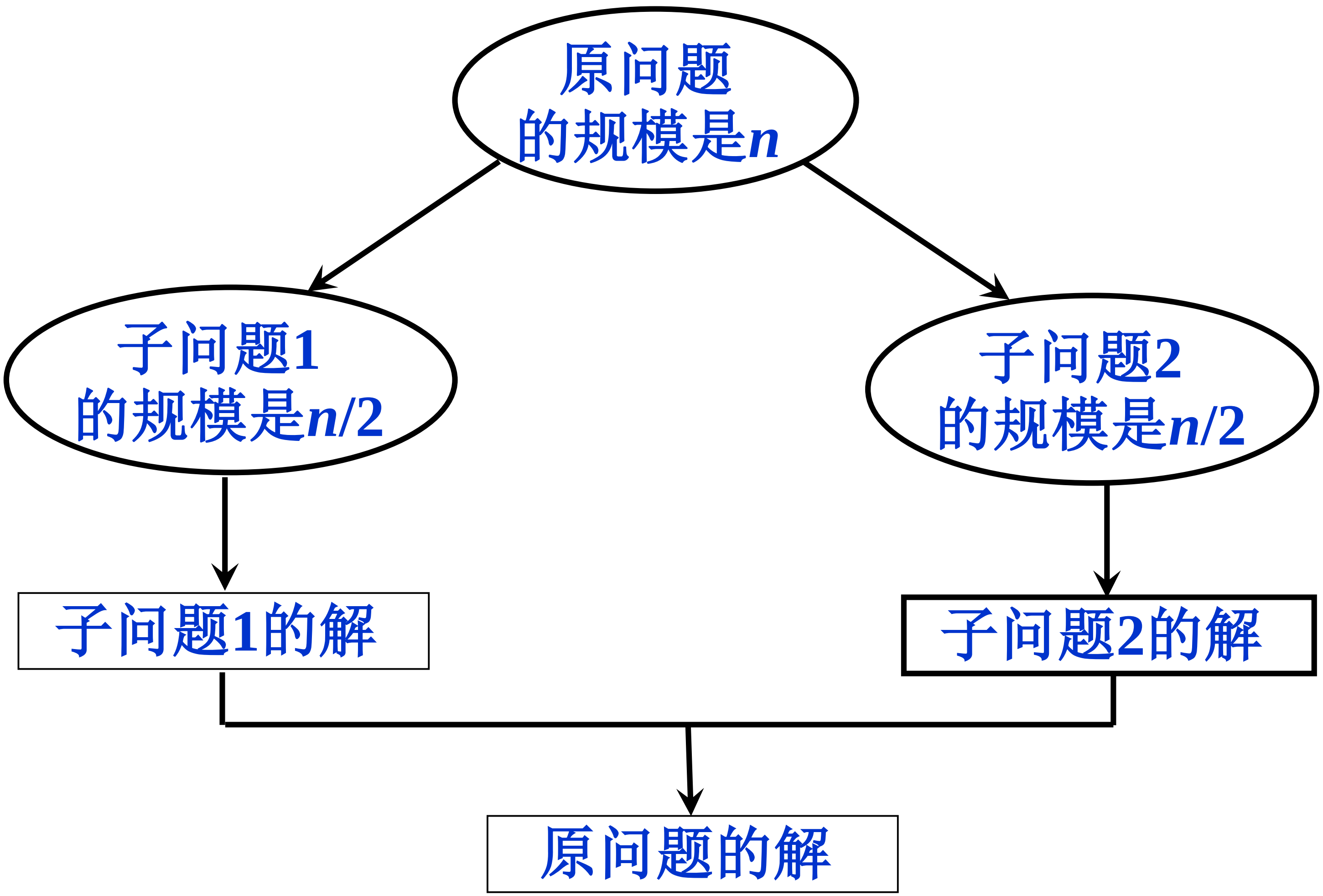
“分治合”策略

算法设计思想

- 合——将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

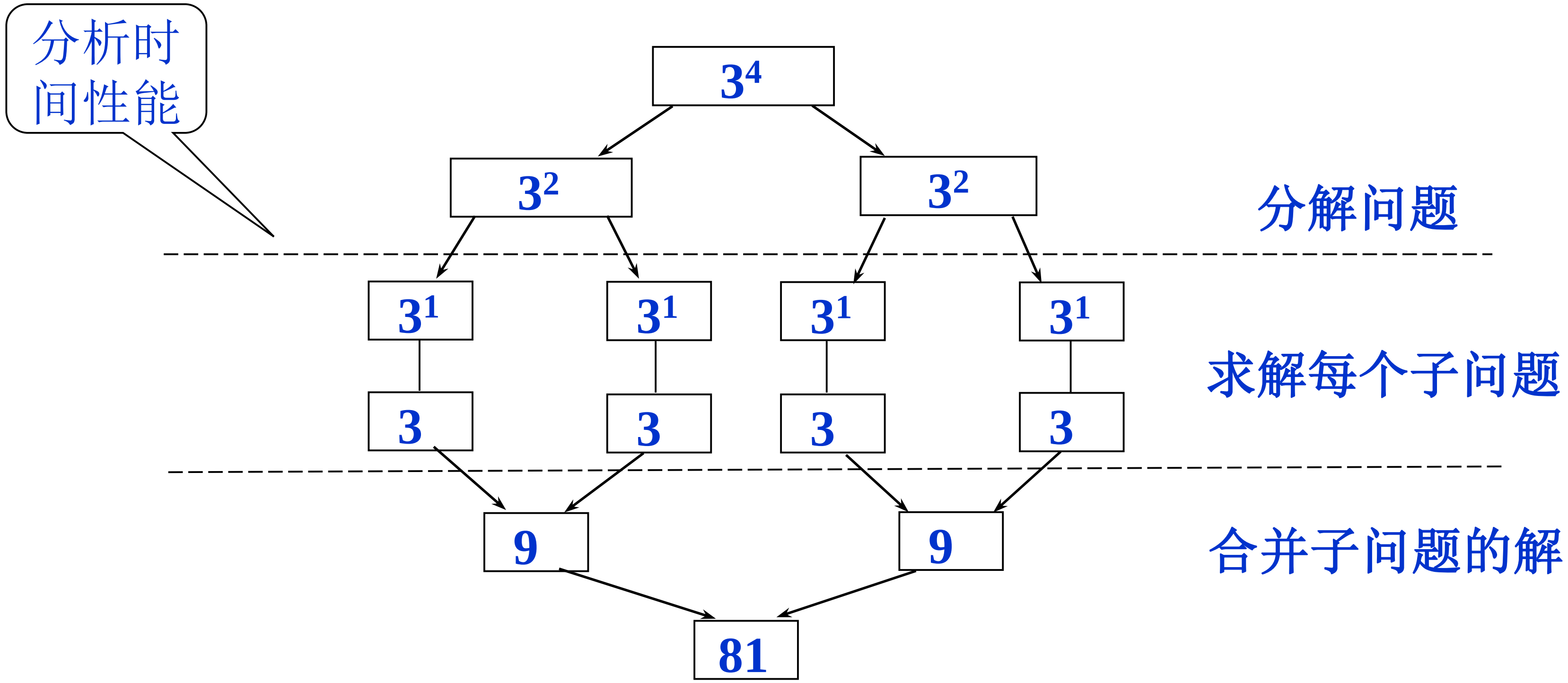


分治法的典型情况



例：计算a^n，应用分治技术得到如下计算方法：

$$a^n = \begin{cases} a & \text{如果 } n = 1 \\ a^{\lfloor n/2 \rfloor} \quad a^{\lceil n/2 \rceil} & \text{如果 } n > 1 \end{cases}$$

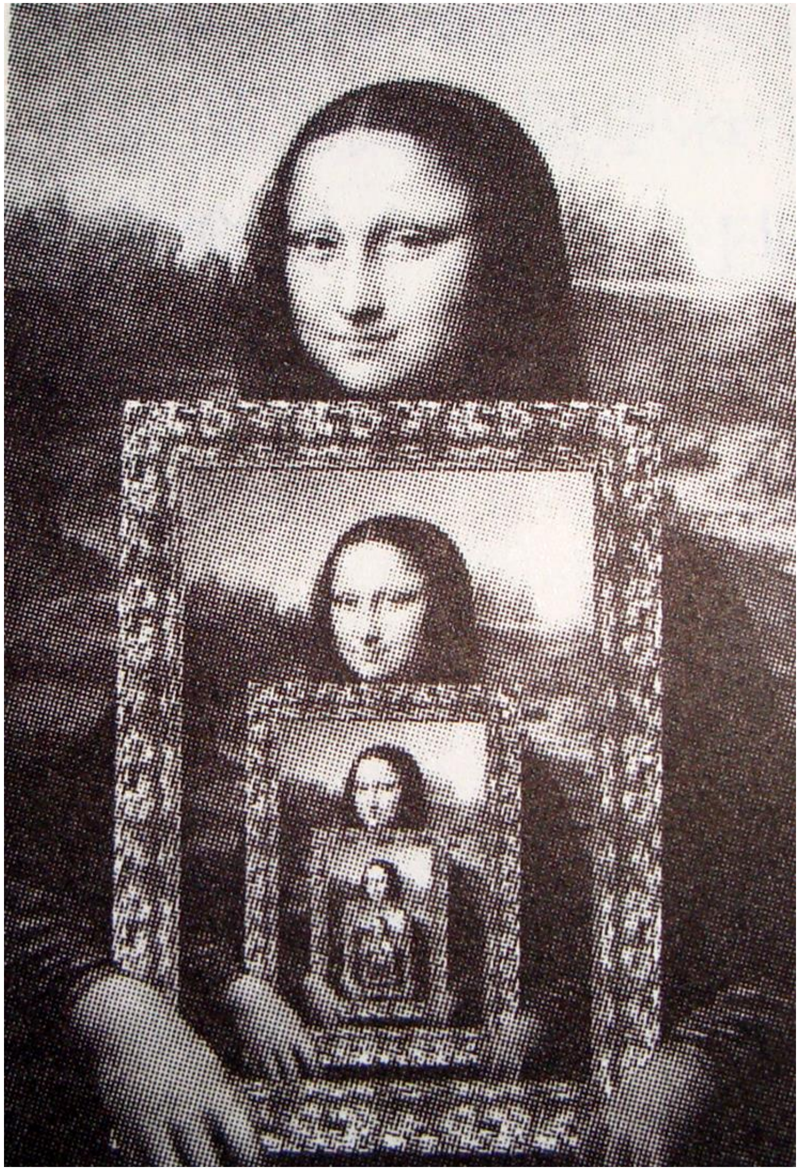


分治与递归

- “你站在桥上看风景，看风景人在楼上看你，明月装饰了你的窗子，你装饰了别人的梦。”

  ------卞（bian)之琳　《断章》

- 这首诗包含的是一个递归概念，如果细心些，你会发现，生活中到处是递归。



递归的蒙娜丽莎

递归的定义

递归（Recursion）就是子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己，是一种描述问题和解决问题的基本方法。

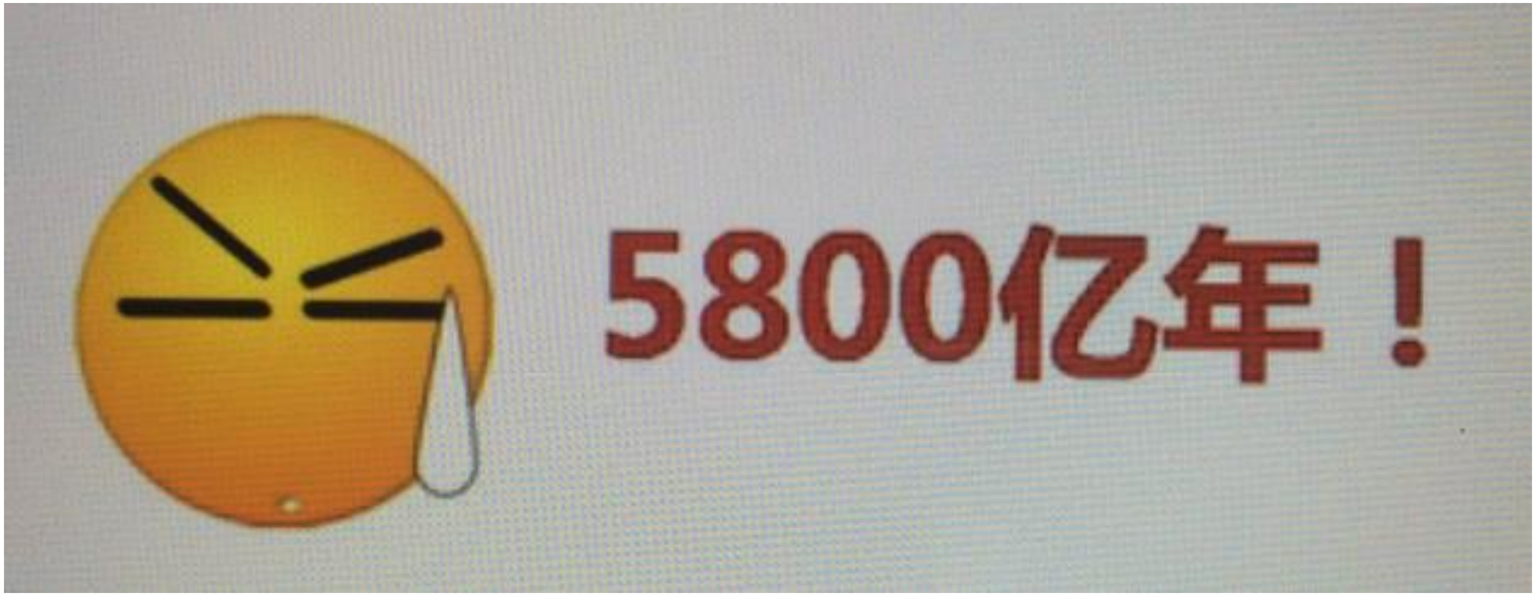
递归有两个基本要素：

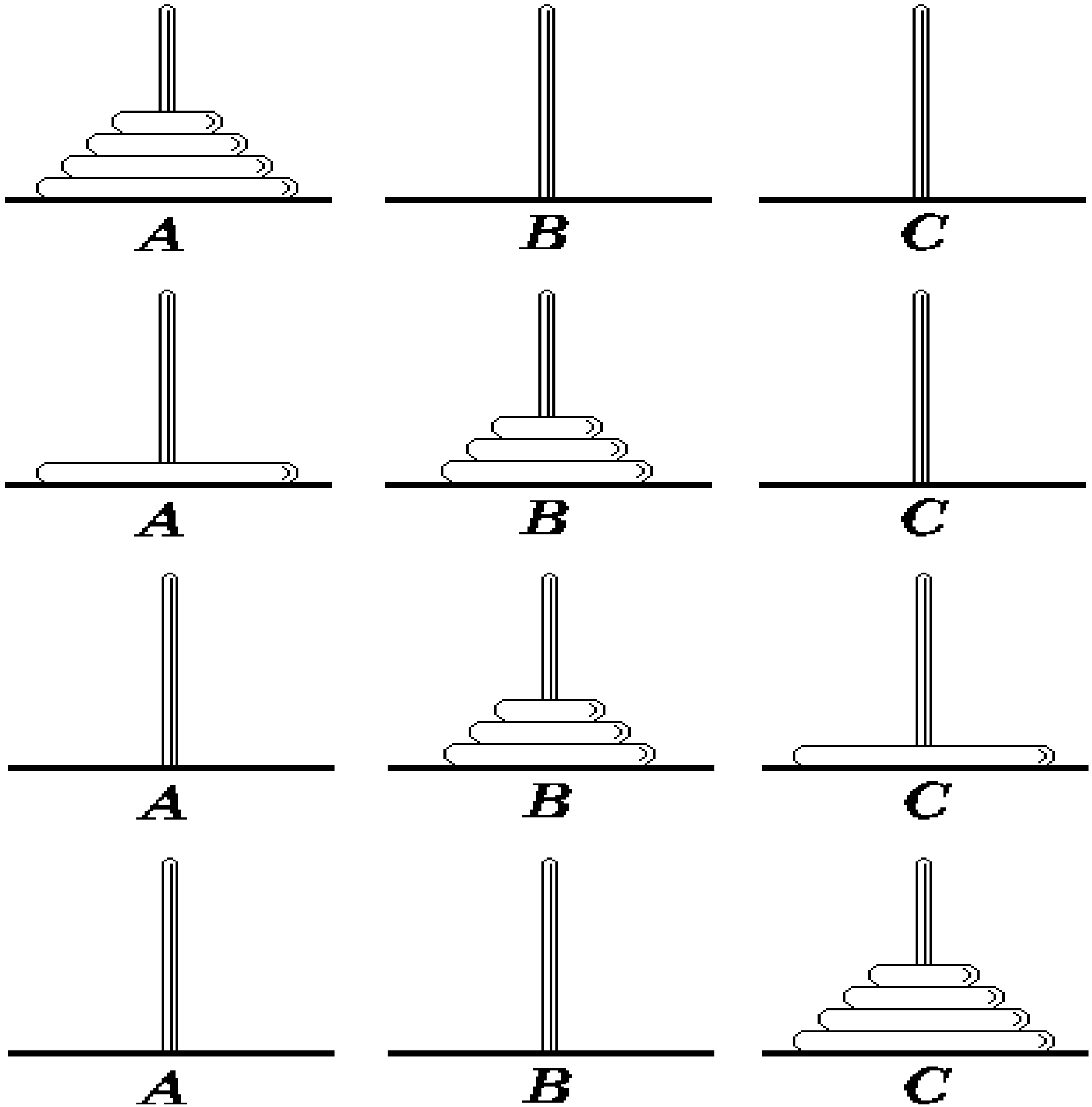
(1) 边界条件：确定递归到何时终止；

(2) 递归模式：大问题是如何分解为小问题的。

递归函数的经典问题——汉诺塔问题

在世界刚被创建的时候有一座钻石宝塔（塔A），其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。


5800亿年！


A
B
C
A
B
C
A
B
C
A
B
C

汉诺塔问题可以通过以下三个步骤实现：

（**1**）将塔**A**上的**n-1**个碟子借助塔**C**先移到塔**B**上。

（**2**）把塔**A**上剩下的一个碟子移到塔**C**上。

（**3**）将**n-1**个碟子从塔**B**借助塔**A**移到塔**C**上。

显然，这是一个递归求解的过程

C++描述

算法——汉诺塔算法

```
void Hanoi(int n, char A, char B, char C)
    //第一列为语句行号
{
    if (n==1) Move(A, C);
    //Move是一个抽象操作，表示将碟子从A移到C上
    else {
        Hanoi(n-1, A, C, B);
        Move(A, C);
        Hanoi(n-1, B, A, C);
    }
}
```

递归算法**结构清晰**，**可读性强**，而且容易用数学归纳法来证明算法的正确性，因此，它为设计算法和调试程序带来很大方便，是算法设计中的一种**强有力的工具**。

分治与递归

- 分治是一种思想，递归是一种手段。

- 关键点：
- 1）如何分解？
- 2）如何合并？

启发式规则：

1. 平衡子问题：最好使子问题的规模大致相同。

2. 独立子问题：各子问题之间相互独立，这涉及到分治法的效率，如果各子问题不是独立的，则分治法需要重复地解公共的子问题。

划分策略

划分策略很关键，总体上可以分为两大类：

1. **黑盒划分**此类方法根据问题的规模对原问题进行划分，而不考虑划分对象的属性值，所以形象地称之为黑盒划分**策略**，；
2. **白盒划分策略**，此方法根据划分对象的特定属性值（也称之为参照值）把对象集合划分为若干个子集。

一、 排序问题中的分治法

1. 归并排序
2. 快速排序

Sorting

- Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

归并排序

【例】归并排序

任务描述：任意给定一包含n个整数的集合，把n个整数按升序排列。

输入：每测试用例包括两行，第一行输入整数个数，第二行输入n个整数，数与数之间用空格隔开。最后一行包含-1，表示输入结束。

输出：每组测试数据的结果输出占一行，输出按升序排列的n个整数。

样例输入：

7

49 38 65 97 76 13 27

-1

样例输出：

13 27 38 49 65 76 97

二路归并排序的分治策略是：

（1）划分：将待排序序列$r_1, r_2, \ldots, r_n$划分为两个长度相等的子序列$r_1, \ldots, r_{n/2}$和$r_{n/2+1}, \ldots, r_n$；

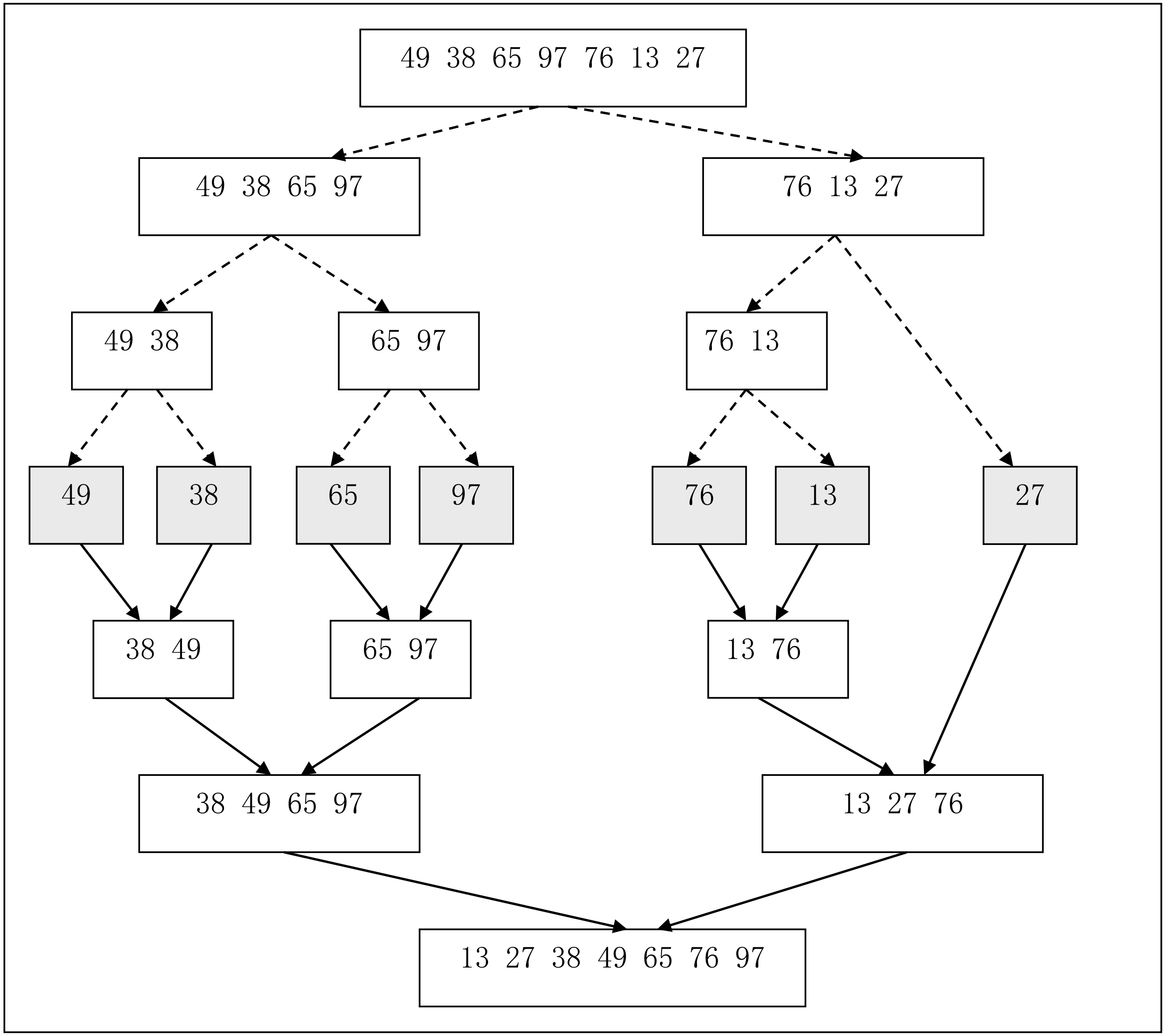
（2）求解子问题：分别对这两个子序列进行排序，得到两个有序子序列；

（3）合并：将这两个有序子序列合并成一个有序序列。

$r_1 \cdots \quad \cdots\ r_{n/2} \mid r_{n/2+1} \cdots \quad \cdots\ r_n$ 划分

$r'_1 < \cdots \quad \cdots < r'_{n/2} \mid r'_{n/2+1} < \cdots \quad \cdots < r'_n$ 递归处理

$r''_1 < \cdots\cdots < r''_{n/2} < r''_{n/2+1} < \cdots\cdots < r''_n$ 合并解

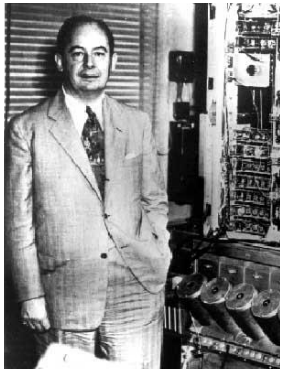

49 38 65 97 76 13 27
49 38 65 97
76 13 27
49 38
65 97
76 13
49
38
65
97
76
13
27
38 49
65 97
13 76
38 49 65 97
13 27 76
13 27 38 49 65 76 97

C++描述

算法——归并排序

```
void MergeSort（int r[ ], int r1[ ], int s, int t）
 {
    if（s= =t）r1[s]=r[s];
      else {
         m=（s+t）/2;
         Mergesort（r, r1, s, m）;    //归并排序前半个子序列
         Mergesort（r, r1, m+1, t）;   //归并排序后半个子序列
         Merge（r1, r, s, m, t）;      //合并两个已排序的子序列
      }
 }
```

- Mergesort.
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.

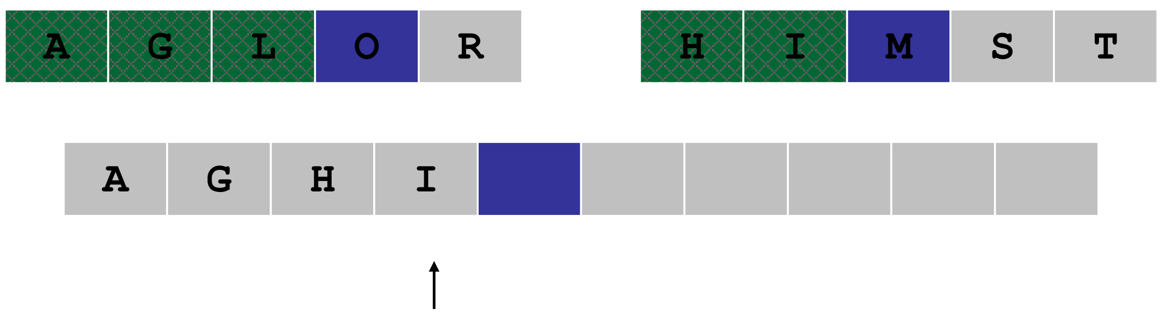


Jon von Neumann (1945)

A L G O R I T H M S			
A L G O R	I T H M S	divide	$O(1)$
A G L O R	H I M S T	sort	$2T(n/2)$
A G H I L M O R S T		merge	$O(n)$

Merging

- Combine two pre-sorted lists into a sorted whole.
- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



using only a constant amount of extra storage

递归表达式求解

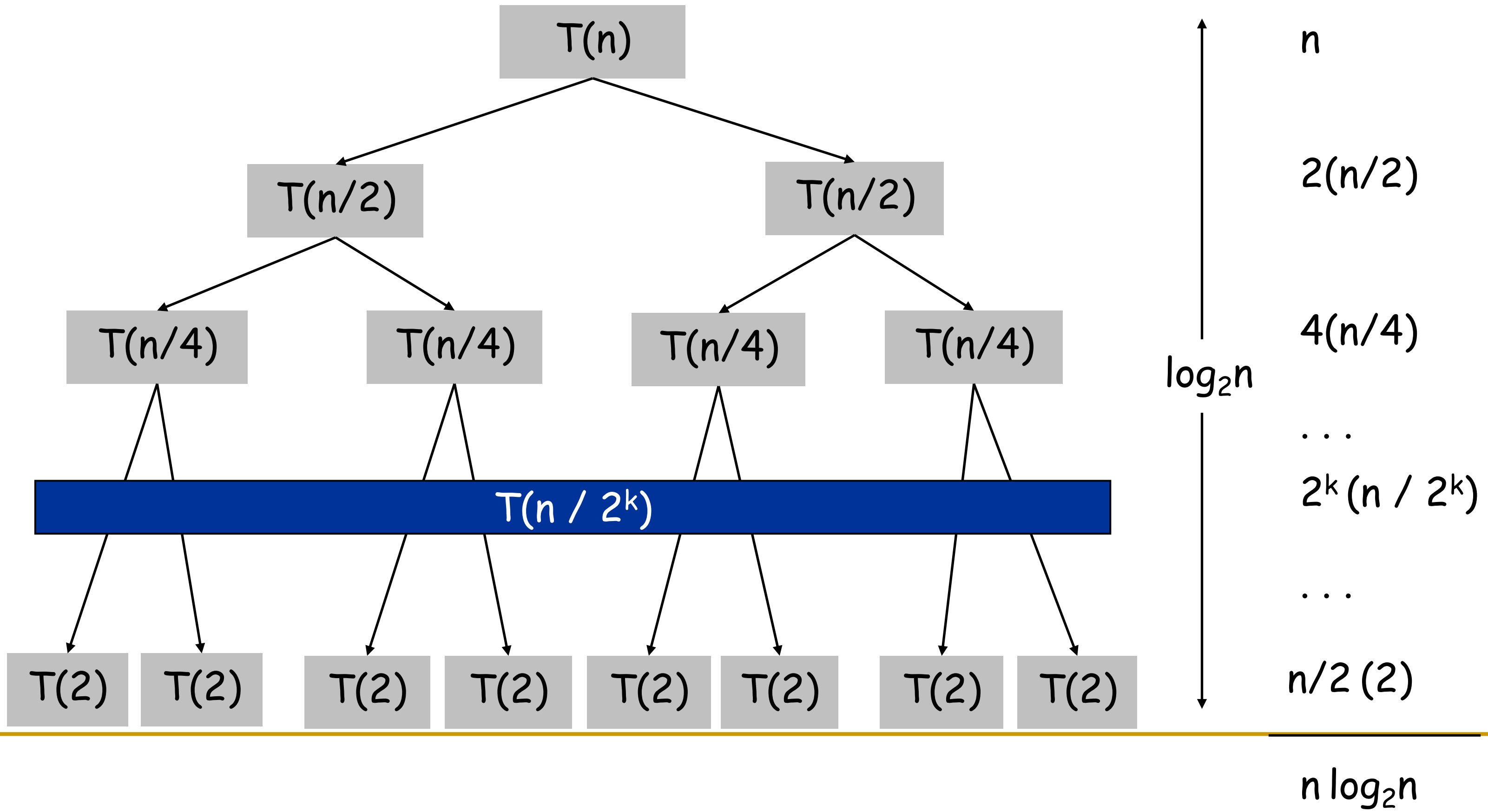
- 求解递归表达式是分析分治策略的重要基础

- 不容易看出里面所隐含的执行序列和规律。但如果用图形的方式展开，就容易得多，这就是---递归树。

1、 Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

对于归并排序，最好、最坏、平均复杂度一致----一视同仁，显著特点！



2. 猜测技术:

对递推关系式估计一个上限，证明（用数学归纳法）

例 使用猜测技术分析二路归并排序算法的时间复杂性。

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

假定 $T(n)\leqslant n^2$，并证明这个猜测是正确的。在证明中，为了计算方便，假定 $n=2^k$。

对于最基本的情况，$T(2)=1\leqslant 2^2$；对于所有 $i\leqslant n$，假设 $T(i)\leqslant i^2$，而

$$T(2n)=2T(n)+2n\leqslant 2n^2+2n\leqslant 4n^2=(2n)^2$$

由此，$T(n)=O(n^2)$成立。

$O(n^2)$是一个最小上限吗？如果猜测更小一些，例如对于某个常数 c，$T(n)\leqslant cn$，很明显这样做不行。所以，真正的代价一定在 n 和 n^2 之间。

现在试一试 $T(n)\leqslant n\log_2 n$。

3、 扩展递归技术

$$T(n)=\begin{cases}7 & n=1\\ 2T(n/2)+5n^2 & n>1\end{cases}$$

$$\begin{aligned}T(n)&=2T(n/2)+5n^2\\&=2(2T(n/4)+5(n/2)^2)+5n^2\\&=2(2(2T(n/8)+5(n/4)^2)+5(n/2)^2)+5n^2\\&=2^kT(1)+2^{k-1}5(\frac{n}{2^{k-1}})^2+\ldots+2\times5(\frac{n}{2})^2+5n^2\end{aligned}$$

$$T(n)=7n+5\sum_{i=0}^{k-1}\left(\frac{n}{2^i}\right)^2=7n+5n^2(2-\frac{1}{2^{k-1}})\;=10n^2-3n\le10n^2=O(n^2)$$

4、 通用分治递推式（大师解法）

大小为*n*的原问题分成若干个大小为*n*/*b*的子问题，其中*a*个子问题需要求解，而cn^k是合并各个子问题的解需要的工作量。

$$T(n)=\begin{cases} c & n=1 \\ aT(n/b)+cn^k & n>1 \end{cases}$$

$$T(n)=\begin{cases} O(n^{\log_b a}) & a>b^k \\ O(n^k \log_b n) & a=b^k \\ O(n^k) & a<b^k \end{cases}$$

4. 通用分治递推式（大师解法）

大小为n的原问题分成若干个大小为n/b的子问题，其中a个子问题需要求解，而cn^k是合并各个子问题的解需要的工作量。

$$T(n)=\begin{cases} c & n=1 \\ aT(n/b)+cn^k & n>1 \end{cases}$$

$$T(n)=\begin{cases} O(n^{\log_b a}) & a>b^k \\ O(n^k \log_b n) & a=b^k \\ O(n^k) & a<b^k \end{cases}$$

逆序对问题

【例】逆序对问题

设$A[1..n]$是一个包含n个非负整数的数组。如果在$i< j$的情况下，有$A[i]>A[j]$，则$(A[i],A[j])$就称为A中的一个**逆序对**。

例如，数组（3，1，4，5，2）的“逆序对”有 <3,1>, <3,2>，<4,2>，<5,2>，共4个。

给定任意非负n维数组，计算其逆序对的数目。

Applications

- Applications.
  - Collaborative filtering（协同过滤）.
  - meta-searching（元搜索） on the Web.
  - etc.

- Brute force: check all $\Theta(n^2)$ pairs i and j.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

<u>Inversions</u>

3-2, 4-2

当数组中元素的个数*n*比较少时，容易统计逆序对的数目

- 数组中只有1个元素的话，逆序对的数目为0；
- 数组中有2个元素，如果第一个元素大于第二个元素，则有1个逆序对，否则0个逆序对。
- 若数组中元素个数超过2，怎么办？

- Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

- Divide-and-conquer.
  - Divide: separate list into two pieces.

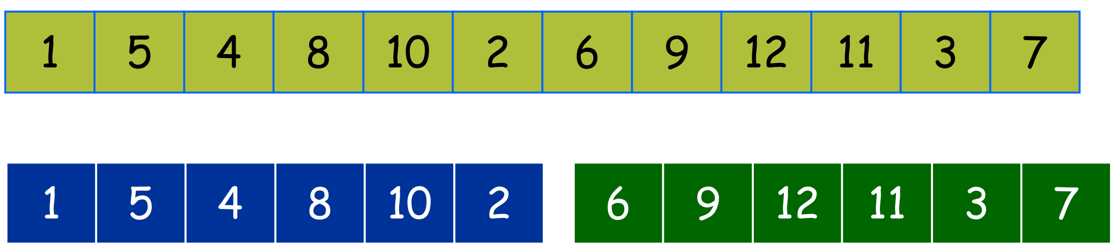
1	5	4	8	10	2	6	9	12	11	3	7

Divide: $O(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.



Divide: O(1).

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

5-4, 5-2, 4-2, 8-2, 10-2

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

- Combine: count blue-green inversions
  - Assume each half is sorted.
  - Count inversions where a_i and a_j are in different halves.
  - Merge two sorted halves into sorted whole.

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)

$$T(n) \le T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + O(n) \quad \Rightarrow \quad \mathrm{T}(n) = O(n \log n)$$

2. 快速排序

快速排序

【例 5-7】快速排序

任务描述： 任意给定一个包含 n 个整数的集合，把这 n 个整数按升序排列。

输入： 每个测试用例包括两行，第一行输入整数的个数 $n, n \le 10000$，第二行输入 n 个整数，

数与数之间用空格隔开。最后一行包含-1，表示输入结束。

输出： 每组测试数据的结果输出占一行，输出按升序排列的 n 个整数。

样例输入：

7

49 38 65 97 76 13 27

-1

样例输出：

13 27 38 49 65 76 97

快速排序的分治策略是：

（1）划分：选定一个记录作为轴值，以轴值为基准将整个序列划分为两个子序列r_1 … r_{i-1}和r_{i+1} … r_n，前一个子序列中记录的值均小于或等于轴值，后一个子序列中记录的值均大于或等于轴值；

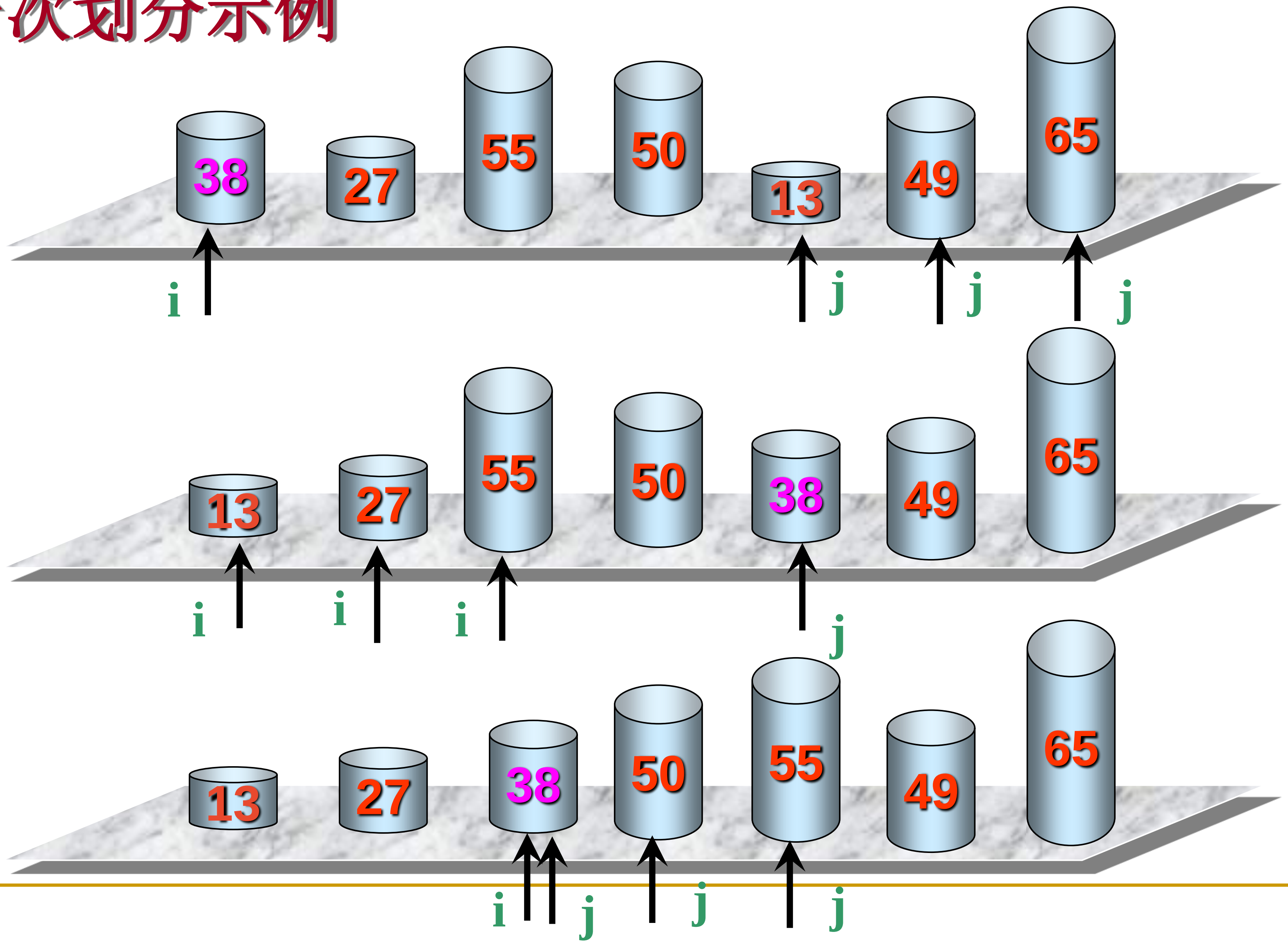
（2）求解子问题：分别对划分后的每一个子序列递归处理；

（3）合并：由于对子序列r_1 … r_{i-1}和r_{i+1} … r_n的排序是就地进行的，所以合并不需要执行任何操作。

$$[\ \underbrace{r_1\ \dots\ \dots\ r_{i-1}}_{\text{均}\le r_i}\ ]\ \underset{\substack{\uparrow\\ \text{轴值}\\ \text{位于最终位置}}}{r_i}\ [\ \underbrace{r_{i+1}\ \dots\ \dots\ r_n}_{\text{均}\ge r_i}\ ]$$

- ❖归并排序按照记录在序列中的位置对序列进行划分，
- ❖快速排序按照记录的值对序列进行划分。

一次划分示例

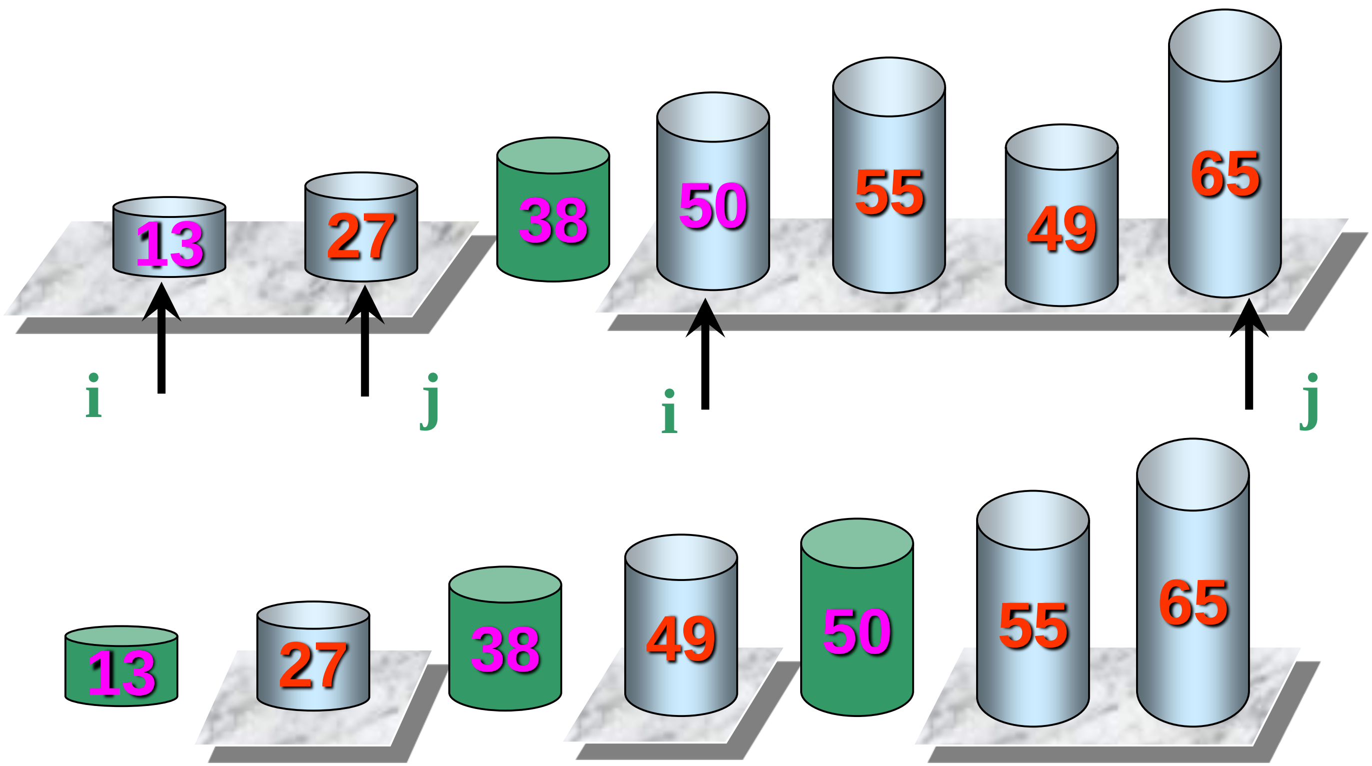

38
27
55
50
13
49
65
i
j
j
j
13
27
55
50
38
49
65
i
i
i
j
13
27
38
50
55
49
65
i
j
j
j

C++描述

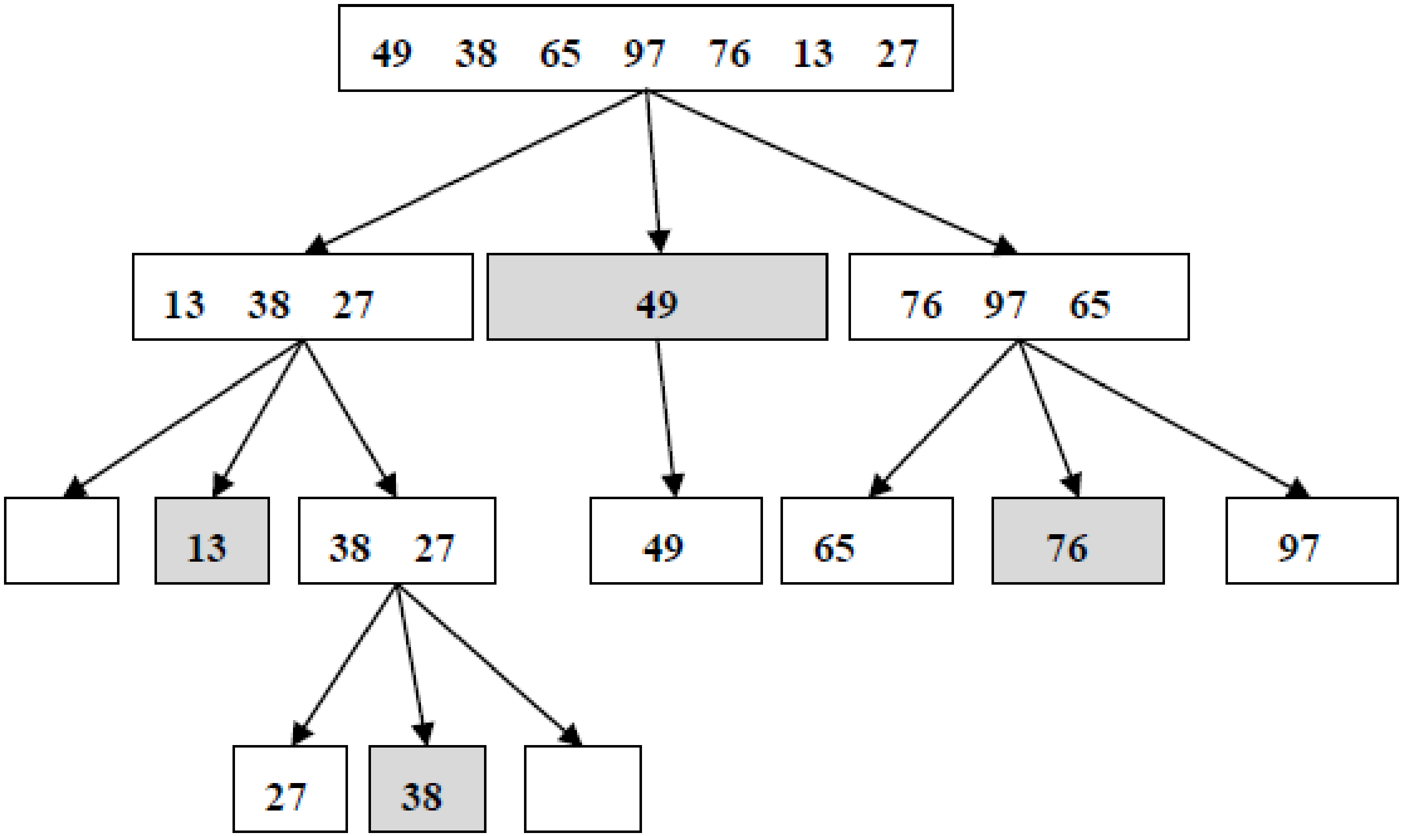
算法4.5——一次划分

```
int Partition（int r[ ], int first, int end）
{
    i=first; j=end;        //初始化
    while（i<j）
    {
        while（i<j && r[i]<= r[j]）j--；   //右侧扫描
         if（i<j）{
            r[i]←→r[j];           //将较小记录交换到前面
            i++;
         }
        while（i<j && r[i]<= r[j]）i++；   //左侧扫描
        if（i<j）{
           r[j]←→r[i];           //将较大记录交换到后面
           j--;
        }
    }
    retutn i;   // i为轴值记录的最终位置
}
```

以轴值为基准将待排序序列划分为两个子序列后，对每一个子序列分别递归进行排序。



快速排序



C++描述

算法4.6——快速排序

```
void QuickSort（int r[ ], int first, int end）
{
   if（first<end）{
      pivot=Partition（r, first, end）;
        //问题分解，pivot是轴值在序列中的位置
      QuickSort（r, first, pivot−1）;
        //递归地对左侧子序列进行快速排序
      QuickSort（r, pivot+1, end）;
       //递归地对右侧子序列进行快速排序
   }
}
```

快速排序的运行时间与划分是否对称有关：

1）最坏情况，对于一个已经排好序的集合（比如{1, 2, 3, 4, 5, 6}），调用快速排序的时间复杂度为：

$$T(n)=\begin{cases} O(1) & n\le 1 \\ T(n-1)+O(n) & n>1 \end{cases}$$

2）最好情况，每次划分所取的基准都恰好为中值，即每次划分都产生两个大小为n/2的区域，时间复杂度为：

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

时间复杂度：分解的次数和每次分解需要比较的次数是决定因素。

在最好情况下，每次划分对一个记录定位后，该记录的左侧子序列与右侧子序列的长度相同。在具有n个记录的序列中，一次划分需要对整个待划分序列扫描一遍，则所需时间为$O(n)$。设$T(n)$是对n个记录的序列进行排序的时间，每次划分后，正好把待划分区间划分为长度相等的两个子序列，则有：

$$T(n) \le 2T(n/2) + n$$

$$\le 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$$

$$\le 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$$

$$\cdots \ \cdots \ \cdots$$

$$\leqslant nT(1) + n\log_2 n = O(n\log_2 n)$$

因此，时间复杂度为$O(n\log_2 n)$。

在最坏情况下，待排序记录序列正序或逆序，每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空）。

$$\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}n(n-1) = O(n^2)$$

因此，时间复杂度为**$O(n^2)$**。

在平均情况下，设基准记录的关键码第***k***小（**1≤*k*≤*n***），则有：

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}(T(n-k)+T(k-1))+n=\frac{2}{n}\sum_{k=1}^{n}T(k)+n$$

这是快速排序的平均时间性能，可以用归纳法证明，其数量级也为$O(n\log_2 n)$。

二. 组合问题中的分治法

1. 最大子段和问题
2. 棋盘覆盖问题

1. 最大子段和问题

给定由n个整数组成的序列(a_1, a_2, …, a_n)，最大子段和问题要求该序列形如$\sum_{k=i}^{j} a_k$的最大值（$1 \le i \le j \le n$），当序列中所有整数均为负整数时，其最大子段和为0。例如，序列(−20, 11, −4, 13, −5, −2)的最大子段和为：

$$\sum_{k=2}^{4} a_k = 20$$

最大子段和问题的分治策略是：

（1）划分：按照平衡子问题的原则，将序列(a_1, a_2, …, a_n)划分成长度相同的两个子序列(a_1, …, $a_{\lfloor n/2 \rfloor}$)和($a_{\lceil n/2 \rceil + 1}$, …, a_n)，则会出现以下三种情况：

① $a_1, \ldots, a_n$的最大子段和＝$a_1, \ldots, a_{\lfloor n/2 \rfloor}$的最大子段和；

② $a_1, \ldots, a_n$的最大子段和＝$a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$的最大子段和；

③ $a_1, \ldots, a_n$的最大子段和＝ $\sum_{k=i}^{j} a_k$ ，且

$$1 \le i \le \lfloor n/2 \rfloor,\ \lfloor n/2 \rfloor + 1 \le j \le n$$

（2）求解子问题：对于划分阶段的情况①和②可递归求解，情况③需要分别计算 $s1 = \max \sum_{k=i}^{\lfloor n/2 \rfloor} a_k \, (1 \le i \le \lfloor n/2 \rfloor)$ $s2 = \max \sum_{k=\lfloor n/2 \rfloor+1}^{j} a_k \, (\lfloor n/2 \rfloor + 1 \le j \le n)$，则$s1+s2$为情况③的最大子段和。

（3）合并：比较在划分阶段的三种情况下的最大子段和，取三者之中的较大者为原问题的解。

$a_1 \ldots \ldots a_i \ldots a_{mid} \,|\, a_{mid+1} \ldots a_j \ldots \ldots a_n$ --------- 划分 $(mid = \lfloor n/2 \rfloor)$

leftsum　　rightsum --------- 递归处理

max{leftsum, sum, rightsum} --------- 合并解

sum --------- 不能递归处理

$a_1 \ldots \ldots a_i \ldots a_{mid} \,|\, a_{mid+1} \ldots a_j \ldots \ldots a_n$ --------- 最大子段和横跨两个子序列

C++描述

算法——最大子段和问题

```
int MaxSum(int a[ ], int left, int right)
{
    sum=0;
    if (left= =right) {     //如果序列长度为1，直接求解
        if (a[left]>0) sum=a[left];
        else sum=0;
    }
    else {
        center=(left+right)/2;    //划分
        leftsum=MaxSum(a, left, center);
                                        //对应情况①，递归求解
        rightsum=MaxSum(a, center+1, right);
                                        //对应情况②，递归求解
```

```
      s1=0; lefts=0;            //以下对应情况③，先求解s1
      for (i=center; i>=left; i--)
      {
         lefts+=a[i];
         if (lefts>s1) s1=lefts;
      }
      s2=0; rights=0;           //再求解s2
      for (j=center+1; j<=right; j++)
      {
         rights+=a[j];
         if (rights>s2) s2=rights;
      }
      sum=s1+s2;                //计算情况③的最大子段和
      if (sum<leftsum) sum=leftsum;
               //合并，在sum、leftsum和rightsum中取较大者
      if (sum<rightsum) sum=rightsum;
   }
   return sum;
}
```

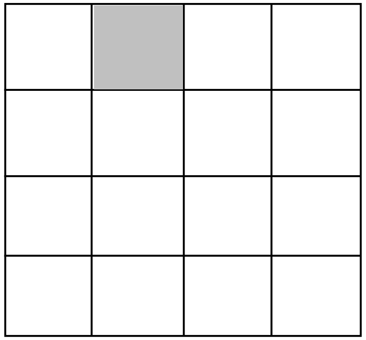
分析算法4.7的时间性能，对应划分得到的情况①和②，需要分别递归求解，对应情况③，两个并列**for**循环的时间复杂性是$O(n)$，所以，存在如下递推式：

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

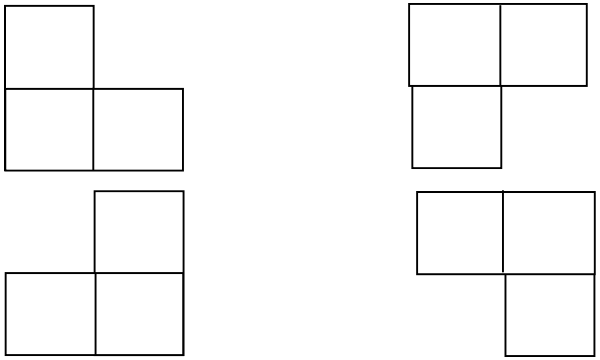
根据1.2.4节主定理，算法4.7的时间复杂性为$O(n\log_2 n)$。

2. 棋盘覆盖问题

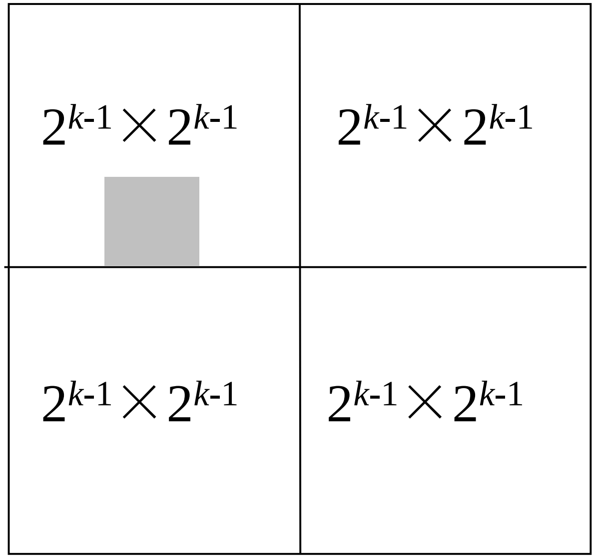
在一个$2^k \times 2^k$（$k \geq 0$）个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为特殊方格。棋盘覆盖问题要求用图所示的4种不同形状的L型骨牌覆盖给定棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。



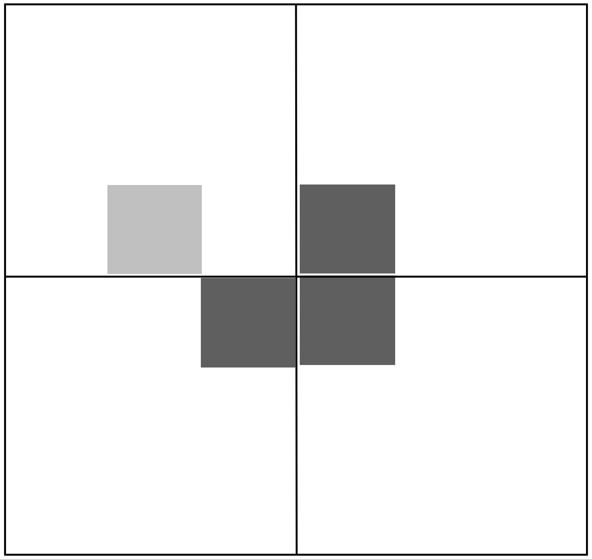
(a) k=2时的一种棋盘



(b) 4种不同形状的L型骨牌



(a)棋盘分割



(b) 构造相同子问题

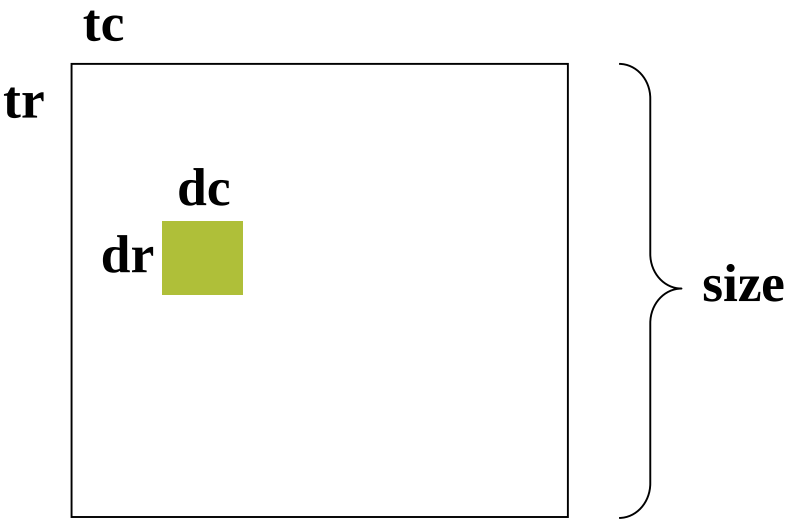
下面讨论棋盘覆盖问题中数据结构的设计。

（1）棋盘：可以用一个二维数组board[size][size]表示一个棋盘，其中，size=2^k。为了在递归处理的过程中使用同一个棋盘，将数组board设为全局变量；

（2）子棋盘：整个棋盘用二维数组board[size][size]表示，其中的子棋盘由棋盘左上角的下标tr、tc和棋盘大小s表示；

（3）特殊方格：用board[dr][dc]表示特殊方格，dr和dc是该特殊方格在二维数组board中的下标;

（4） L型骨牌：一个2^k×2^k的棋盘中有一个特殊方格，所以，用到L型骨牌的个数为$(4^k-1)/3$，将所有L型骨牌从1开始连续编号，用一个全局变量t表示。



棋盘覆盖问题中的数据结构

C++描述

算法4.8——棋盘覆盖

```
void  ChessBoard(int tr, int tc, int dr, int dc, int size)
// tr和tc是棋盘左上角的下标，dr和dc是特殊方格的下标，
// size是棋盘的大小，t已初始化为0
{
     if (size = = 1) return;  //棋盘只有一个方格且是特殊方格
     t++;  // L型骨牌号
     s = size/2;  // 划分棋盘
     // 覆盖左上角子棋盘
     if (dr < tr + s && dc < tc + s)   // 特殊方格在左上角子棋盘中
        ChessBoard(tr, tc, dr, dc, s);          //递归处理子棋盘
     else{      // 用 t 号L型骨牌覆盖右下角，再递归处理子棋盘
       board[tr + s - 1][tc + s - 1] = t;
       ChessBoard(tr, tc, tr+s-1, tc+s-1, s);
     }
```

```
// 覆盖右上角子棋盘
if (dr < tr + s && dc >= tc + s)    // 特殊方格在右上角子棋盘中
  ChessBoard(tr, tc+s, dr, dc, s);         //递归处理子棋盘
else {       // 用 t 号L型骨牌覆盖左下角，再递归处理子棋盘
  board[tr + s - 1][tc + s] = t;
  ChessBoard(tr, tc+s, tr+s-1, tc+s, s); }
// 覆盖左下角子棋盘
if (dr >= tr + s && dc < tc + s)   // 特殊方格在左下角子棋盘中
  ChessBoard(tr+s, tc, dr, dc, s);        //递归处理子棋盘
else {      // 用 t 号L型骨牌覆盖右上角，再递归处理子棋盘
  board[tr + s][tc + s - 1] = t;
  ChessBoard(tr+s, tc, tr+s, tc+s-1, s); }
// 覆盖右下角子棋盘
if (dr >= tr + s && dc >= tc + s)  // 特殊方格在右下角子棋盘中
  ChessBoard(tr+s, tc+s, dr, dc, s);      //递归处理子棋盘
else {      // 用 t 号L型骨牌覆盖左上角，再递归处理子棋盘
  board[tr + s][tc + s] = t;
  ChessBoard(tr+s, tc+s, tr+s, tc+s, s); }
}
```

2	2	3	3	7	7	8	8
2	1	1	3	7	6	6	8
4	1	5	5	9	9	6	10
4	4	5	0		9	10	10
12	12	13	0	0	17	18	18
12	11	13	13	17	17	16	18
14	11	11	15	19	16	16	20
14	14	15	15	19	19	20	20

设$T(k)$是覆盖一个$2^k \times 2^k$棋盘所需时间，从算法的划分策略可知，$T(k)$满足如下递推式：

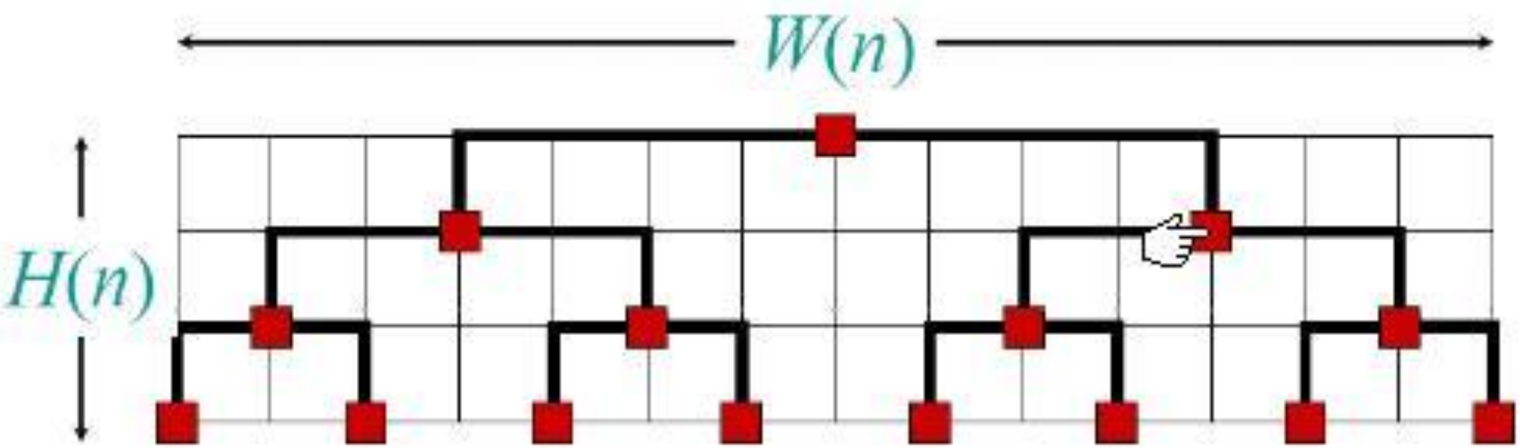
$$T(k)=\begin{cases} O(1) & k=0 \\ 4T(k-1)+O(1) & k>0 \end{cases}$$

解此递推式可得$T(k)=O(4^k)$。由于覆盖一个$2^k \times 2^k$棋盘所需的骨牌个数为$(4^k-1)/3$，所以，算法4.8是一个在渐进意义下的最优算法。

VLSI布线问题

- 布线是集成电路设计中一个重要的方面

基本方案



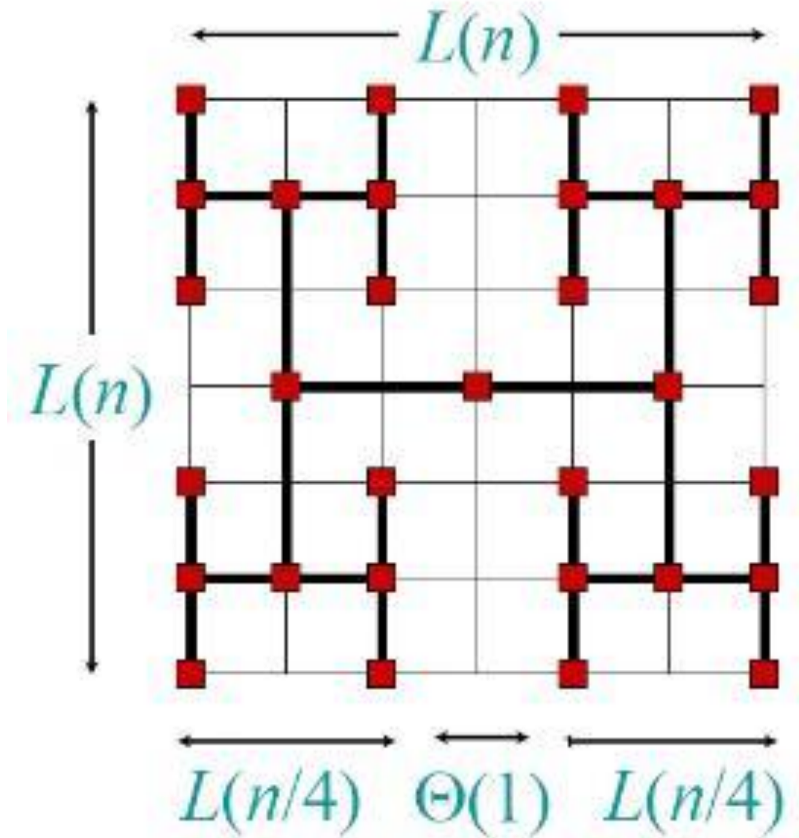
$$H(n) = H(n/2) + \Theta(1) = \Theta(\lg n)$$

$$W(n) = 2W(n/2) + \Theta(1) = \Theta(n)$$

$$\text{Area} = \Theta(n \lg n)$$

这是最优的布局吗？

H型嵌入



$$L(n) = 2L(n/4) + \Theta(1)$$
$$= \Theta(\sqrt{n})$$

$$\text{Area} = \Theta(n)$$

总结归纳

- 分治是一种解题的策略，基本思想是：“如果整个问题比较复杂，可以将问题分化，各个击破。”
- 分治包含“分”和“治”两层含义，如何分，分后如何“治”成为解决问题的关键所在
- 不是所有的问题都可以采用分治
- 分治可进行二分，三分等等，具体怎么分，需看问题的性质和分治后的效果。
- 只有深刻地领会分治的思想，认真分析分治后可能产生的预期效率，才能灵活地运用分治思想解决实际问题。

分治就在潜意识深处

- 分治并不是一个特定的算法，而是一个算法思想。这种思想来源于人们偏向处理简单的事情，因为简单的东西比复杂的东西好处理。
- 与分治不可分割的一个概念就是递归，没有递归，分治也就无从落地，而成了空中楼阁。因此，如何递归就成为分治策略的根基。分治与递归互为因果。

练习

- 假如你正在为一投资公司咨询。他们正在做模拟，对一给定的股票连续观察n天，记为i=1,2,…,n;对每天i,该股票每股的价格p(i)。假设在这个时间区间内，在某一天他们想买1000（第i天）股而在另一天（第j天）卖出所有这些股。为得到最多收益，他们应什么时候买什么时候卖？请设计算法在O(nlogn)时间内找到正确的i与j.

思考题

- 问题描述：

在一个按照东西和南北方向划分成规整街区的城市里，n 个居民点散乱地分布在不同的街区中。用x 坐标表示东西向，用y 坐标表示南北向。各居民点的位置可以由坐标(x,y)表示。街区中任意2 点(x 1,y 1)和(x2,y2)之间的距离可以用数值|x1-x2|+|y 1-y2|度量居民们希望在城市中选择建立邮局的最佳位置，使n 个居民点到邮局的距离总和最小。

编程任务：

给定n 个居民点的位置,编程计算n 个居民点到邮局的距离总和的最小值。