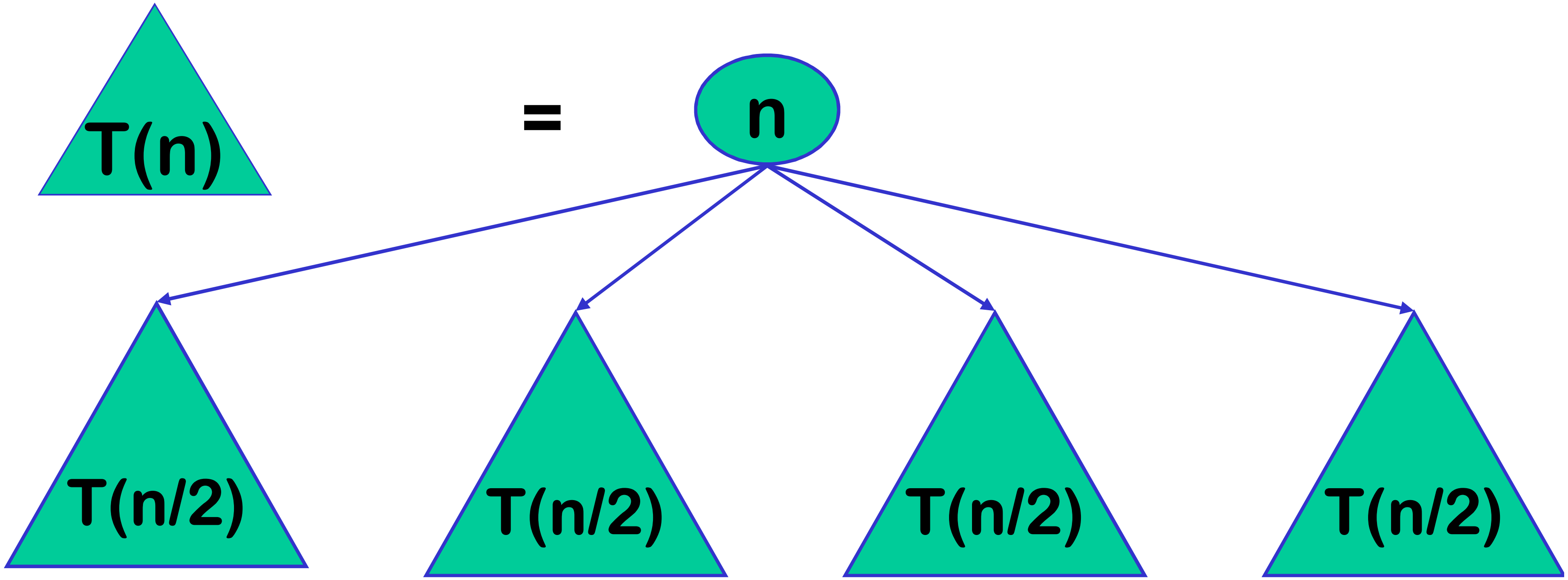
第2章　递归与分治策略

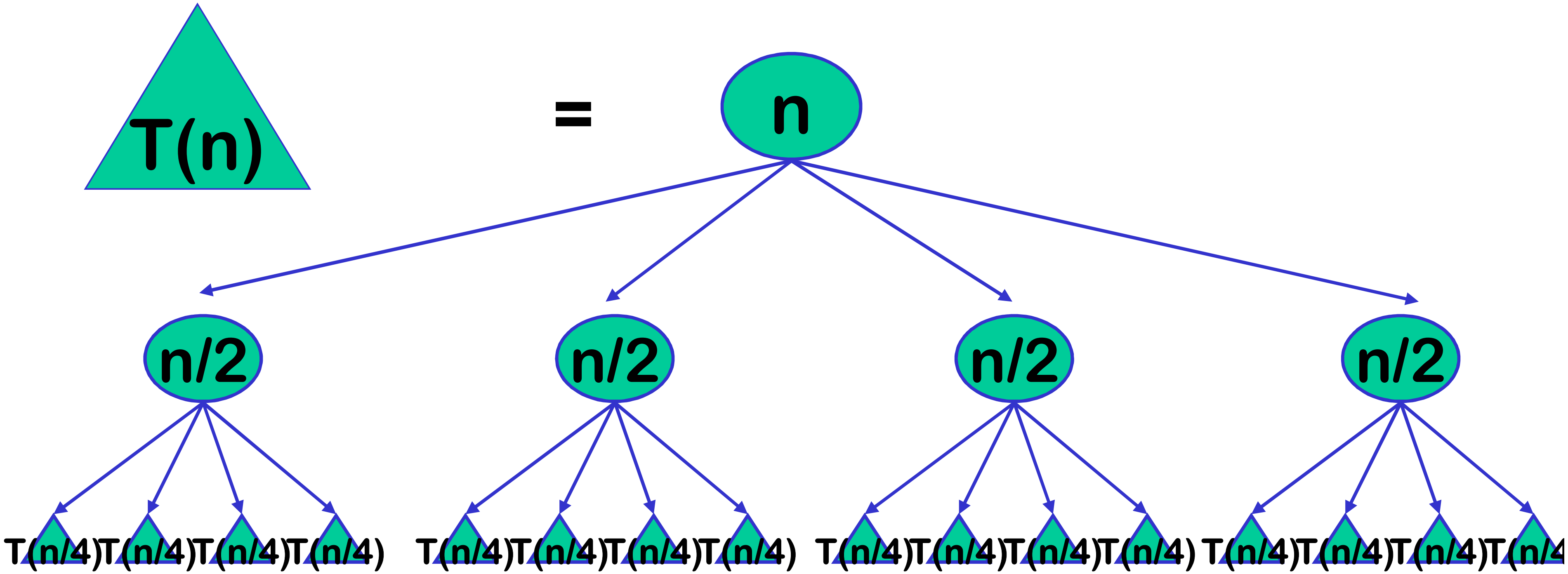
算法总体思想

- 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。



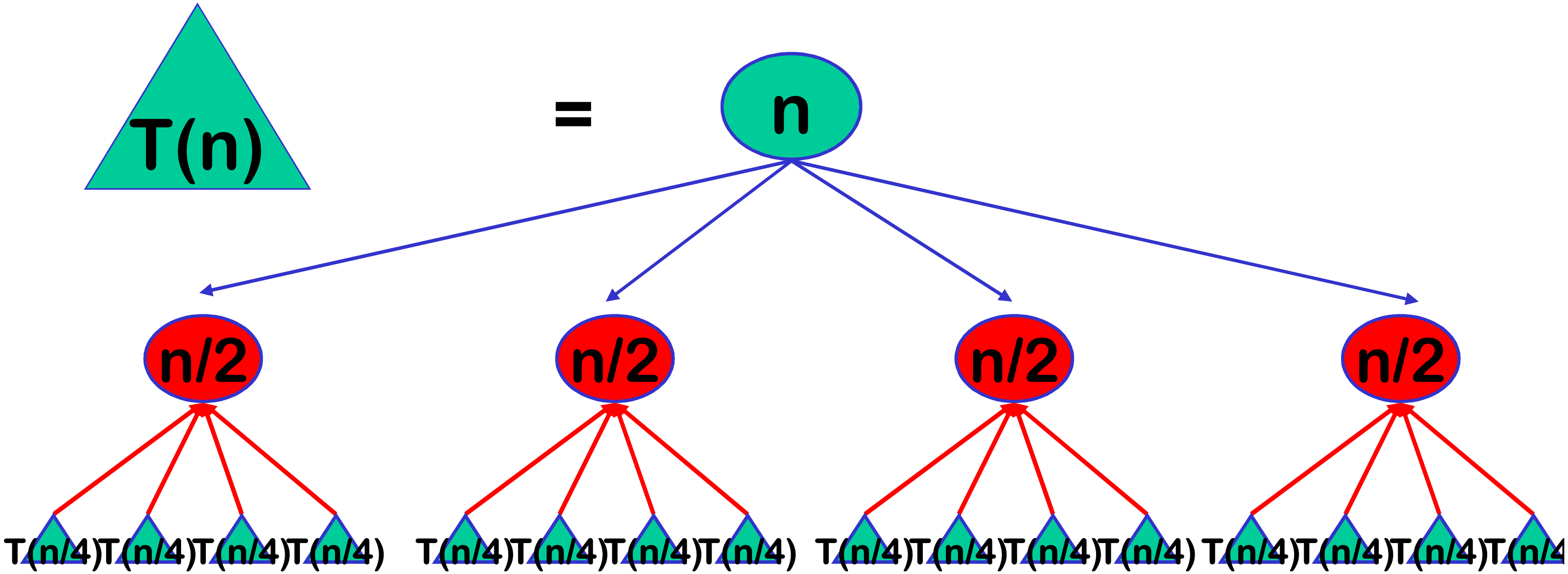
算法总体思想

- 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。



算法总体思想

- 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。



算法总体思想

- 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。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

凡治众如治寡，分数是也。

----孙子兵法

2.1 递归的概念

- 直接或间接地调用自身的算法称为递归算法。用函数自身给出定义的函数称为递归函数。
- 由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。
- 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。

下面来看几个实例。

2. 1　递归的概念

例1　阶乘函数

阶乘函数可递归地定义为：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

边界条件

递归方程

边界条件与递归方程是递归函数的二个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

2.1 递归的概念

例2 Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，…，被称为Fibonacci数列。它可以递归地定义为：

$$F(n)=\begin{cases}1 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1\end{cases}$$

边界条件

递归方程

第n个Fibonacci数可递归地计算如下：

```
public static int fibonacci(int n)
  {
      if (n <= 1) return 1;
      return fibonacci(n-1)+fibonacci(n-2);
  }
```

2.1　递归的概念

例3　Ackerman函数

当一个函数及它的一个变量是由函数自身定义时，称这个函数是双递归函数。

Ackerman函数**A(n，m)**定义如下：

$$
\begin{cases}
A(1,0)=2 & \\
A(0,m)=1 & m\geq 0\\
A(n,0)=n+2 & n\geq 2\\
A(n,m)=A(A(n-1,m),m-1) & n,m\geq 1
\end{cases}
$$

2.1 递归的概念

例3 Ackerman函数

前2例中的函数都可以找到相应的非递归方式定义：

$$n! = 1 \cdot 2 \cdot 3 \cdots\cdots (n-1) \cdot n$$

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

但本例中的Ackerman函数却无法找到非递归的定义。

2.1　递归的概念

例3 Ackerman函数

- A(n，m)的自变量m的每一个值都定义了一个单变量函数：
- M=0时，A(n,0)=n+2
- M=1时，A(n,1)=A(A(n-1,1),0)=A(n-1,1)+2，和A(1,1)=2故A(n,1)=2*n
- M=2时，A(n,2)=A(A(n-1,2),1)=2A(n-1,2)，和A(1,2)=A(A(0,2),1)=A(1,1)=2，故A(n,2)= 2^n 。
- M=3时，类似的可以推出$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$
- M=4时，A(n,4)的增长速度非常快，以至于没有适当的数学式子来表示这一函数。

2.1　递归的概念

例3 Ackerman函数

- 定义单变量的Ackerman函数A(n)为，A(n)=A(n，n)。
- 定义其拟逆函数α(n)为：α(n)=min{k | A(k)≥n}。即α(n)是使n≤A(k)成立的最小的k值。
- α(n)在复杂度分析中常遇到。对于通常所见到的正整数n，有**α(n)≤4**。但在理论上α(n)没有上界，随着n的增加，它以难以想象的慢速度趋向正无穷大。

2.1 递归的概念

例4 排列问题

设计一个递归算法生成n个元素$\{r_1,r_2,\ldots,r_n\}$的全排列。

设$R=\{r_1,r_2,\ldots,r_n\}$是要进行排列的n个元素，$R_i=R-\{r_i\}$。集合X中元素的全排列记为perm(X)。(r_i)perm(X)表示在全排列perm(X)的每一个排列前加上前缀得到的排列。R的全排列可归纳定义如下：

当n=1时，perm(R)=(r)，其中r是集合R中唯一的元素；
当n>1时，perm(R)由(r_1)perm(R_1)，(r_2)perm(R_2)，…，(r_n)perm(R_n)构成。

2.1　递归的概念

例5　整数划分问题

将正整数n表示成一系列正整数之和：$n=n_1+n_2+\ldots+n_k$，其中$n_1\geq n_2\geq\ldots\geq n_k\geq 1$，$k\geq 1$。

正整数n的这种表示称为正整数n的划分。求正整数n的不同划分个数。

例如正整数6有如下11种不同的划分：

6;
5+1;
4+2，4+1+1;
3+3，3+2+1，3+1+1+1;
2+2+2，2+2+1+1，2+1+1+1+1;
1+1+1+1+1+1。

2.1 递归的概念

例5 整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

(3) $q(n,n)=1+q(n,n-1)$;

正整数n的划分由$n_1=n$的划分和$n_1\le n-1$的划分组成。

(4) $q(n,m)=q(n,m-1)+q(n-m,m), n>m>1$;

正整数n的最大加数n_1不大于m的划分由$n_1=m$的划分和$n_1\le n-1$ 的划分组成。

2.1 递归的概念

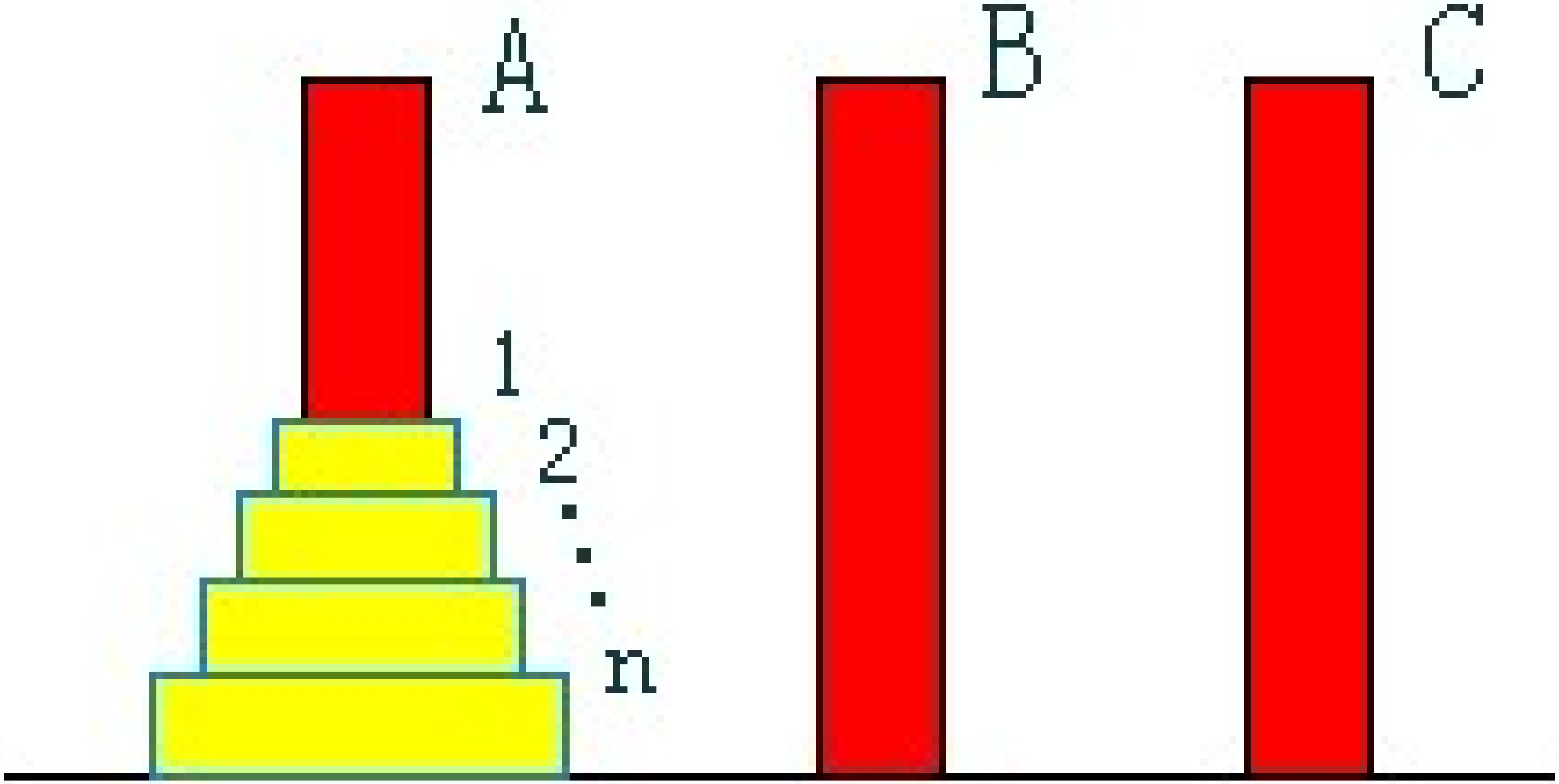
例5 整数划分问题

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设p(n)为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作q(n,m)。可以建立q(n,m)的如下递归关系。

$$
q(n,m)=\begin{cases} 1 & n=1,m=1 \\ q(n,n) & n<m \\ 1+q(n,n-1) & n=m \\ q(n,m-1)+q(n-m,m) & n>m>1 \end{cases}
$$

正整数n的划分数p(n)=q(n,n)。


A
B
C
1
2
n

2.1　递归的概念

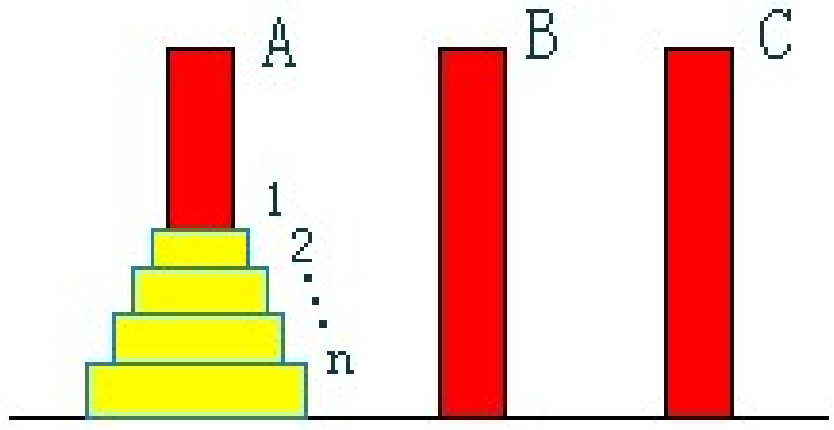
例6　Hanoi塔问题

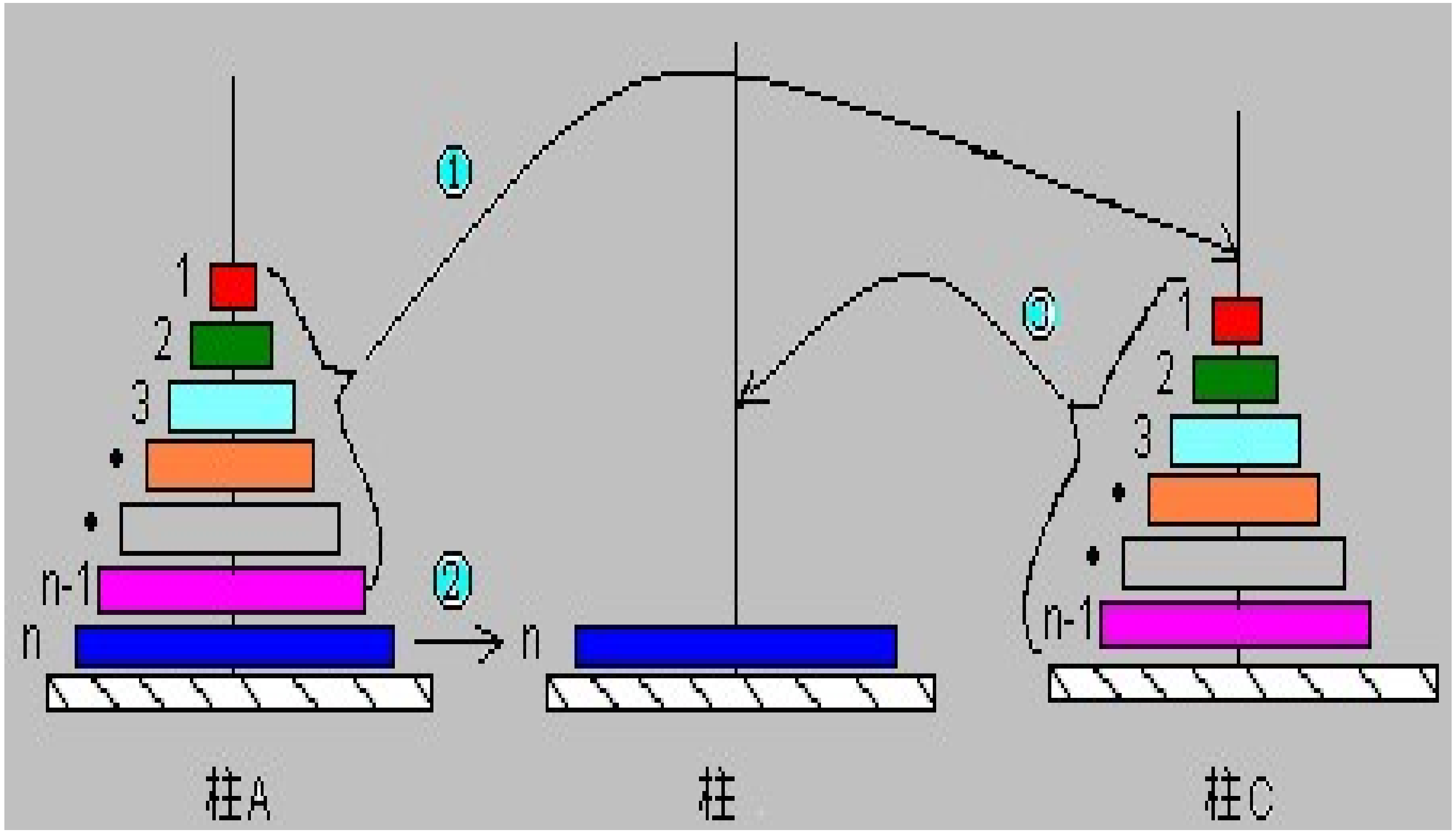
设a,b,c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1,2,…,n,现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

规则1：每次只能移动1个圆盘；

规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；

规则3：在满足移动规则1和2的前提下，可将圆盘移至a,b,c中任一塔座上。



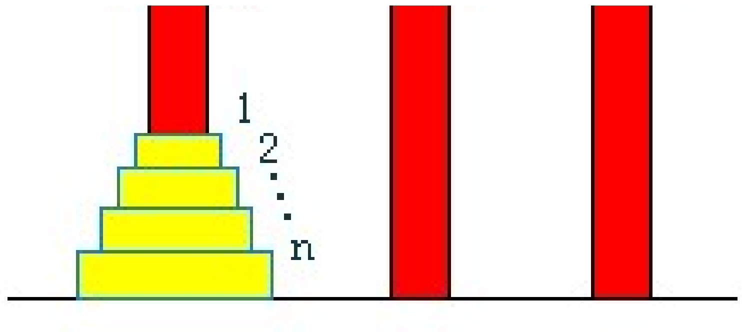


1
2
3
n-1
n
n
n-1
①
②
③
柱A
柱
柱C

2.1　递归的概念

例6　Hanoi塔问题

```
public static void hanoi(int n, int a, int b, int c)
  {
      if (n > 0)
      {
         hanoi(n-1, a, c, b);
         move(a,b);
         hanoi(n-1, c, b, a);
      }
  }
```



2.1 递归小结

优点：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、调试程序带来很大方便。

缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。

2.1 递归小结

解决方法：在递归算法中消除递归调用，使其转化为非递归算法。

1. 采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。
2. 用递推来实现递归函数。
3. 通过Cooper变换、反演变换能将一些递归转化为尾递归，从而迭代求出结果。

后两种方法在时空复杂度上均有较大改善，但其适用范围有限。

2.2 分治法的基本思想
——分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- **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，即该问题具有**最优子结构性质
- **利用该问题分解出的子问题的解可以合并为该问题的解；**
- **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。**

这条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然也可用分治法，但一般用动态规划较好。

2.2 分治法的基本思想
——分治法的基本步骤

```
divide-and-conquer(P)
 {
  if ( | P | <= n0) adhoc(P);   //解决小规模的问题
  divide P into smaller subinstances P1,P2,...,Pk；  //分解问题
  for (i=1,i<=k,i++)
    yi=divide-and-conquer(Pi);  //递归的解各子问题
  return merge(y1,...,yk);  //将各子问题的解合并为原问题的解
 }
```

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。即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。这种使子问题规模大致相等的做法是出自一种平衡(**balancing**)子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n / m的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，下面根据分治模式估计时间复杂度T(n)。

分治法的复杂性分析

```
divide-and-conquer(P)  {
   if ( | P | <= 1) return adhoc(P);
   divide P into smaller subinstances P1,P2,...,Pk;
   for (i=1,i<=k,i++)
     yi=divide-and-conquer(Pi);
   return merge(y1,...,yk);
 }
```

分解为k个子问题以及k个子问题的解合并为原问题的解需用f(n)个单位时间

$$T(n) = \begin{cases} O(1) & n = 1 \\ kT(n/m) + f(n) & n > 1 \end{cases}$$

分治法的复杂性分析

由计算公式

$$T(n)=\begin{cases} O(1) & n=1 \\ kT(n/m)+f(n) & n>1 \end{cases}$$

通过迭代法求得方程的解：

$$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$$

一般算法的复杂性分析

当要解决一个问题时，通常有几个合适的可供选择的算法。我们当然要选择最好的一个。

这就引发了一个问题：如何决定哪个算法是最合适的？

如果只有一个简单问题的一两个实例要解决，那就不必太关心使用哪个算法。此时选一个更容易编程的甚至程序已经有的就可以了。

但如果有很多实例要解决，或者问题相当难解决，那么就需要小心地挑选算法。

一般算法的复杂性分析

算法的挑选有经验研究法和理论研究法两种。

所谓**经验研究法**就是对几种算法分别编程，然后借助计算机，用不同的实例来进行试验，然后选出适合的那个。

所谓**理论研究法**就是以数学化的方式确定算法所需要的资源（计算时间和存储空间）数量与实例大小之间的函数关系，然后根据结果选择算法。通常更关心算法的时间效率。

在今后的学习中采用理论研究法。

一般算法的复杂性分析

不变性原理告诉我们，相同算法的不同实现在效率上不会超过某个常数倍。即只要利用计算机实现一个算法，不管用什么编程语言，用什么编译器，用什么机器运行，程序员用什么技巧（只要不修改算法），运行时间都不会超过某个常数倍。

例如对于规模为n的问题，一种实现运行时花费时间为t1(n)，另一种实现的运行时间为t2(n)，那么当n足够大时，总存在常数c和d，满足

$$c \cdot t2(n) \le t1(n) \le d \cdot t2(n)$$

正是因为改变算法的实现最多得到效率的倍数提升，而改变算法却可能得到效率的指数提升，所以我们更关心算法本身体现的效率——算法的基本运算次数。

一般算法的复杂性分析

一般来说，当问题规模N→∞时，算法的效率T(N)（称为算法的复杂性）也会趋于∞，如果存在G(N)，使得G(N)是与T(N)同阶的无穷大，我们就说T(N)渐进于G(N)，此时就可以用G(N)来作为算法的复杂性度量，称G(N)为渐进复杂性。

例如当$T(N)=3N^2+4N\log N+7$时，取$G(N)=3N^2$，则G(N)是与T(N)同阶的无穷大，因此渐进复杂性G(N)就可以作为算法复杂性的度量。

分析算法的复杂性的目的在于比较两个不同算法的效率，而当两个算法的渐进复杂性的阶不同时，只要能确定出各自的阶，就可以判断哪个算法的效率高。因此渐进复杂性分析只要关心G(N)的阶就够了，这就使复杂性分析得以简化。

一般算法的复杂性分析

在复杂性分析中，常用以下渐进意义下的记号。

设f(N)和g(N)是定义在正数集上的正函数

如果存在正的常数C，当N足够大时，有$f(N) \le Cg(N)$，则称f(N)当N充分大时上有界，g(N)是它的一个**上界**，记为

$$\mathbf{f(N)=O(g(N))}$$

如果存在正的常数C，当N足够大时，有$f(N) \ge Cg(N)$，则称f(N)当N充分大时下有界，g(N)是它的一个**下界**，记为

$$\mathbf{f(N)=\Omega(g(N))}$$

当$f(N)=O(g(N))$且$f(N)=\Omega(g(N))$时，称f(N)与g(N)**同阶**，记为

$$\mathbf{f(N)=\theta(g(N))}$$

2.2 分治法的基本思想 ——分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n / m的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，则有：

$$T(n)=\begin{cases} O(1) & n=1 \\ kT(n/m)+f(n) & n>1 \end{cases}$$

通过迭代法求得方程的解：$T(n)=n^{\log_m k}+\sum_{j=0}^{\log_m n-1} k^j f(n/m^j)$

注意：递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当$m^i \le n < m^{i+1}$时，$T(m^i) \le T(n) < T(m^{i+1})$。

2.3 二分搜索技术

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。

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。**

分析：很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

关于二分搜索（在序列中找到99）

{ 78，87，88，89，90，91，92，93，99，100 }

第一次“二分”，10 / 2 ＝ 5，跟第5个比较（数组下表从1开始）

{ 78，87，88，89，90，91，92，93，99，100 }

99比90大，因而左边无需再找，右边作为整体继续查找。
第二次“二分”，（6+10）/ 2 ＝ 8，跟第8个比较。

{ 78，87，88，89，90，91，92，93，99，100 }

由于99比93大，因而左边无需再找，右边作为整体继续查找。
第三次“二分”，（9+10）/ 2 ＝ 9.5，跟第9个比较。

78，87，88，89，90，91，92，93，99 ，100

2.3 二分搜索技术

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。

据此容易设计出二分搜索算法：

```
public static int binarySearch(int [] a, int x, int n)
  {
   // 在 a[0] <= a[1] <= ... <= a[n-1] 中搜索 x
   // 找到x时返回其在数组中的位置，否则返回-1
   int left = 0; int right = n - 1;
   while (left <= right) {
     int middle = (left + right)/2;
     if (x == a[middle]) return middle;
     if (x > a[middle]) left = middle + 1;
     else right = middle - 1;
    }
   return -1; // 未找到x
  }
```

算法复杂度分析：
每执行一次算法的while循环， 待搜索数组的大小减少一半。因此，在最坏情况下，while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 大整数的乘法

请设计一个有效的算法，可以进行两个n位大整数的乘法运算

◆小学的方法：O(n^2)　　✘效率太低

◆分治法:

X =

Y =

复杂度分析

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

T(n)=O(n^2) **✘没有改进☹**

$X = a\,2^{n/2} + b \qquad Y = c\,2^{n/2} + d$

$XY = ac\,2^n + (ad+bc)\,2^{n/2} + bd$

2. 4 大整数的乘法

请设计一个有效的算法，可以进行两个n位大整数的乘法运算

- ◆小学的方法：$O(n^2)$　　　✘效率太低
- ◆分

XY

复杂度分析

$$T(n)=\begin{cases} O(1) & n=1 \\ 3T(n/2)+O(n) & n>1 \end{cases}$$

$T(n)=O(n^{\log 3})=O(n^{1.59})$ ✓**较大的改进☺**

1. $XY = ac\ 2^n + ((a-b)(d-c)+ac+bd)\ 2^{n/2} + bd$
2. $XY = ac\ 2^n + ((a+b)(d+c)-ac-bd)\ 2^{n/2} + bd$

细节问题：两个XY的复杂度都是$O(n^{\log 3})$，但考虑到a+c,b+d可能得到m+1位的结果，使问题的规模变大，故不选择第2种方案。

2.4 大整数的乘法

请设计一个有效的算法，可以进行两个n位大整数的乘法运算

- ◆小学的方法：$O(n^2)$ ✘效率太低
- ◆分治法: $O(n^{1.59})$ ✓较大的改进
- ◆更快的方法??

➢如果将大整数分成更多段，用更复杂的方式把它们组合起来，将有可能得到更优的算法。

➢最终的，这个思想导致了快速傅利叶变换(Fast Fourier Transform)的产生。该方法也可以看作是一个复杂的分治算法，对于大整数乘法，它能在**O(nlogn)**时间内解决。

➢是否能找到线性时间的算法？？？目前为止还没有结果。

2.5 Strassen矩阵乘法

◆传统方法：O(n^3)

A和B的乘积矩阵C中的元素C[i,j]定义为:$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$

若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C的 个元素所需的计算时间为O(n^3)

2.5 Strassen矩阵乘法

◆传统方法：$O(n^3)$

◆分治法:

使用与 成4

个大小

复杂度分析

$$T(n)=\begin{cases} O(1) & n=2 \\ 8T(n/2)+O(n^2) & n>2 \end{cases}$$

$T(n)=O(n^3)$ ✘**没有改进**☹

由此可得:

$$C_{11}=A_{11}B_{11}+A_{12}B_{21}$$

$$C_{12}=A_{11}B_{12}+A_{12}B_{22}$$

$$C_{21}=A_{21}B_{11}+A_{22}B_{21}$$

$$C_{22}=A_{21}B_{12}+A_{22}B_{22}$$

2.5 Strassen矩阵乘法

◆传统方法：$O(n^3)$
◆分治法:
为了降

复杂度分析

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

$T(n)=O(n^{\log 7}) =O(n^{2.81})$ ✓ **较大的改进☺**

$$M_1 = A_1$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$\Rightarrow$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

2.5 Strassen矩阵乘法

◆传统方法：$O(n^3)$
◆分治法: $O(n^{2.81})$
◆更快的方法??

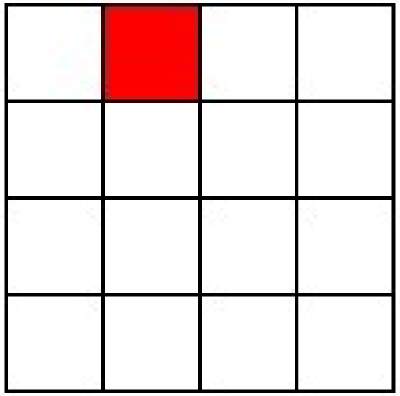
➢Hopcroft和Kerr已经证明(1971)，计算2个 2×2 矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究 3×3 或 5×5 矩阵的更好算法。

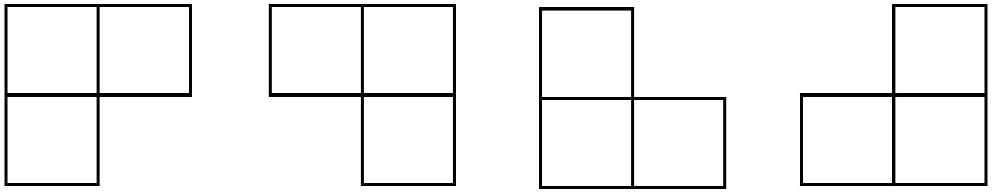
➢在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是 **$O(n^{2.376})$**

➢是否能找到$O(n^2)$的算法？？？目前为止还没有结果。

2.6 棋盘覆盖

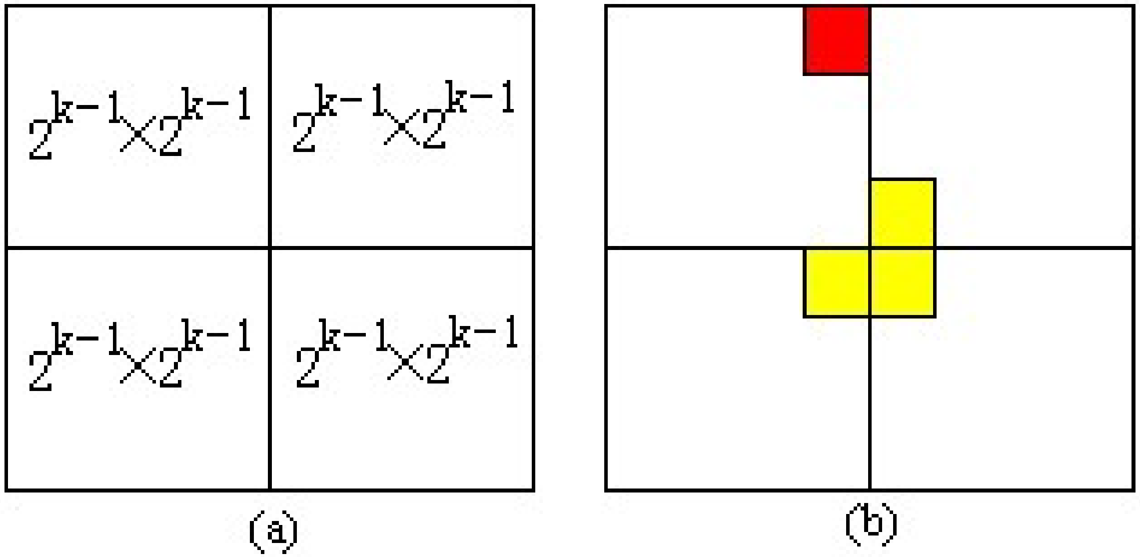
在一个$2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。





2. 6 棋盘覆盖

当$k>0$时，将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$ 子棋盘(a)所示。特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如 (b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘1×1。



2. 6 棋盘覆盖

```
public void chessBoard(int tr, int tc, int dr, int dc, int size)
 {
   if (size == 1) return;
   int t = tile++,  // L型骨牌号
     s = size/2;  // 分割棋盘
   // 覆盖左上角子棋盘
   if (dr < tr
     // 特殊
     chess
   else {// 此
     // 用 t 号
     board[tr
     // 覆盖其余方格
     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);}
   // 覆盖左下角子棋盘
   ...
     s);}
   // 覆盖右下角子棋盘
   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 {// 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);}
 }
```

复杂度分析

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

$T(n)=O(4^k)$ 渐进意义下的最优算法

2.7 合并排序

基本思想：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的

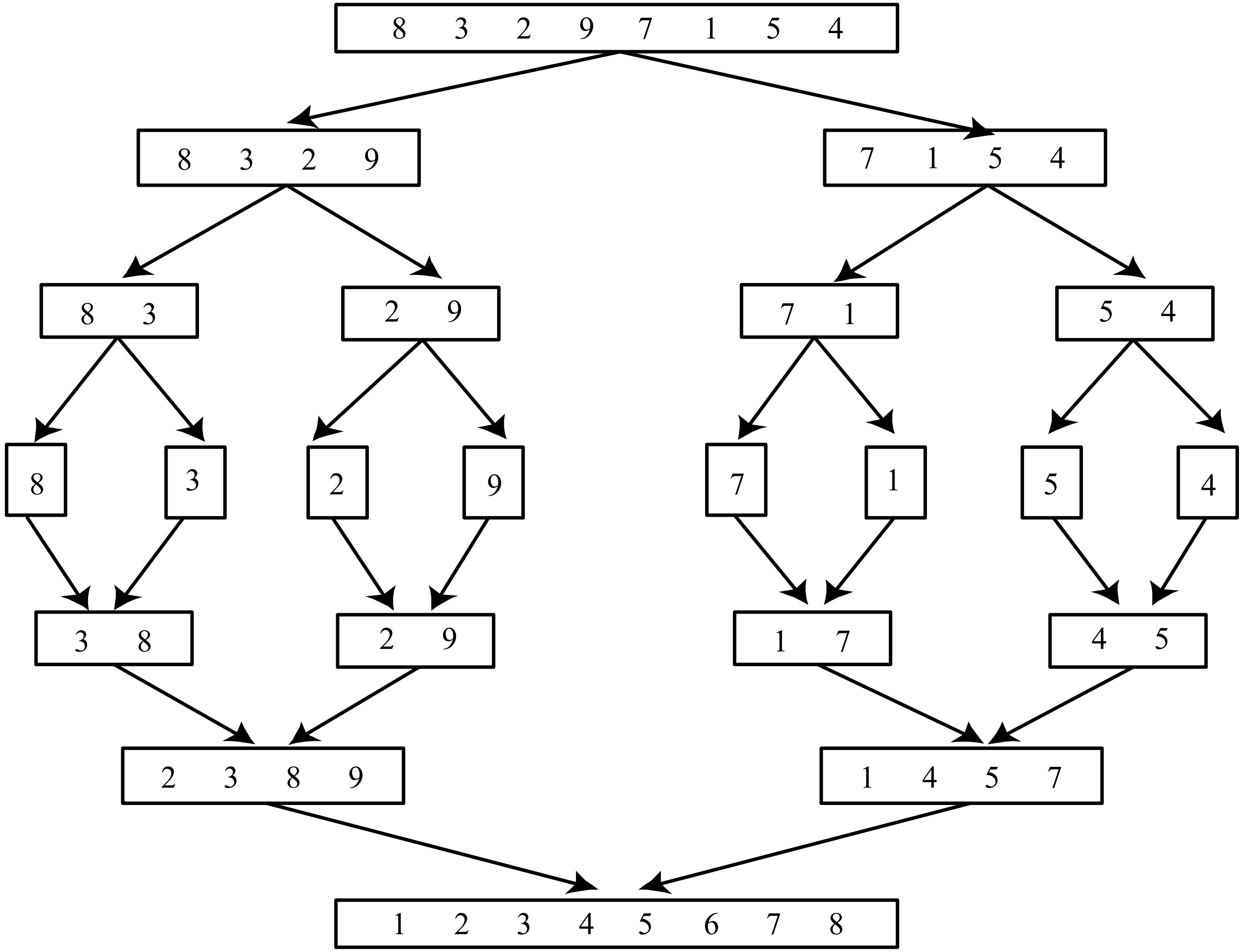
复杂度分析

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

T(n)=O(nlogn) 渐进意义下的最优算法

```
public                                              ght)
{
    if (
    int i=(left+right)/2;  //取中点
    mergeSort(a, left, i);
    mergeSort(a, i+1, right);
    merge(a, b, left, i, right);  //合并到数组b
    copy(a, b, left, right);    //复制回数组a
    }
}
```

合并排序



8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 6 7 8

2.7 合并排序

算法**mergeSort**的递归过程可以消去。

初始序列 [49] [38] [65] [97] [76] [13] [27]

第一步 [38 49] [65 97] [13 76] [27]

第二步 [38 49 65 97] [13 27 76]

第三步 [13 27 38 49 65 76 97]

2.7 合并排序

- 最坏时间复杂度：O(nlogn)
- 平均时间复杂度：O(nlogn)
- 辅助空间：O(n)
- 稳定性：稳定

2.8 快速排序

快速排序是对气泡排序的一种改进方法

它是由C.A.R. Hoare于1962年提出的

在快速排序中，记录的比较和交换是从两端向中间进行的，关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少。

```
private static void qSort(int p, int r)
  {
     if (p<r) {
       int q=partition(p,r); //以a[p]为基准元素将a[p:r]划分成3段a[p:q-1],a[q]和a[q+1:r]，使得a[p:q-1]中任何元素小于等于a[q]，a[q+1:r]中任何元素大于等于a[q]。下标q在划分过程中确定。
       qSort (p,q-1); //对左半段排序
       qSort (q+1,r); //对右半段排序
       }
  }
```

2.8 快速排序

```
private static int partition (int p, int r)
  {
     int i = p,
         j = r + 1;
     Comparable x = a[p];
     // 将>= x的元素交换到左边区域
     // 将<= x的元素交换到右边区域
     while (true) {
       while (a[++i].compareTo(x) < 0);
       while (a[--j].compareTo(x) > 0);
       if (i >= j) break;
       MyMath.swap(a, i, j);
       }
     a[p] = a[j];
     a[j] = x;
     return j;
  }
```

{6, 7, 5, 2, $\bar{5}$, 8}　初始序列

{6, 7, 5, 2, $\bar{5}$, 8}　**j--;**
i　j

{$\bar{5}$, 7, 5, 2, 6, 8}　**i++;**
i　j

{$\bar{5}$, 6, 5, 2, 7, 8}　**j--;**
i　j

{$\bar{5}$, 2, 5, 6, 7, 8}　**i++;**
i　j

{$\bar{5}$, 2, 5} 6 {7, 8}　完成

快速排序具有不稳定性。

2.8 快速排序

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

- **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**

```
private static int randomizedPartition (int p, int r)
  {
    int i = random(p,r);
    MyMath.swap(a, i, p);
    return partition (p, r);
  }
```

2.9 线性时间选择

给定线性序集中n个元素和一个整数k，1≤k≤n，要求找出这n个元素中第k小的元素

```
private static Comparable randomizedSelect(int p,int r,int k)
  {
     if (p==r) return a[p];
     int i=randomizedpartition(p,r),
         j=i-p+1;
     if (k<=j) return randomizedSelect(p,i,k);
     else return randomizedSelect(i+1,r,k-j);
  }
```

在最坏情况下，算法**randomizedSelect**需要$O(n^2)$计算时间但可以证明，算法**randomizedSelect**可以在O(n)平均时间内找出n个输入元素中的第k小元素。