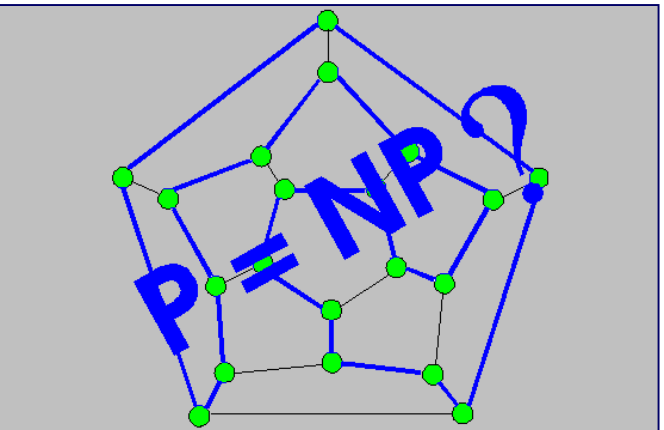
4-分治法

陆伟

算法设计与分析

Introduction to the Design and Analysis of Algorithms



September 21, 2022

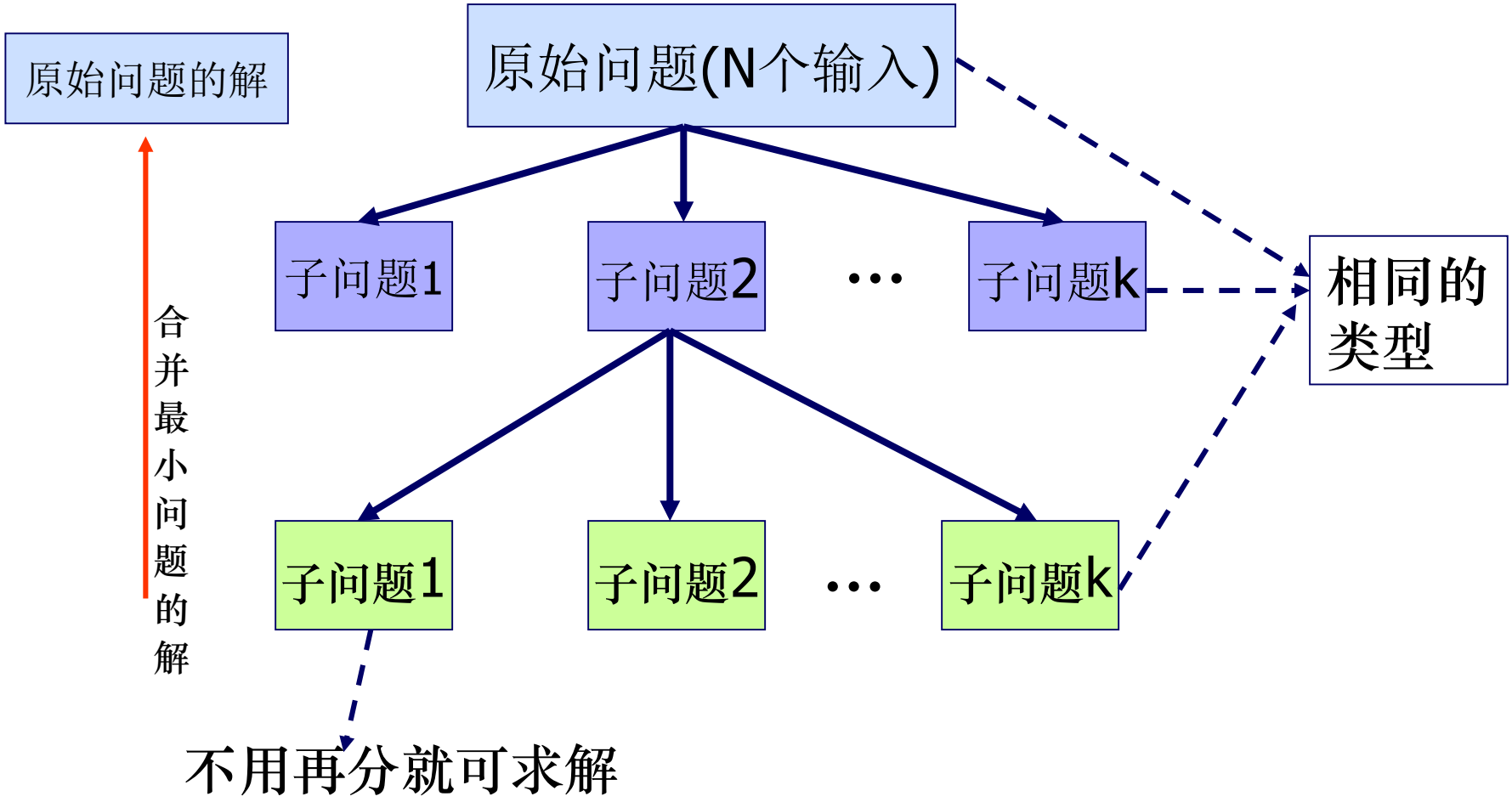
Lecture Overview

1. 基本思想
2. 应用的先决条件
3. 基本步骤
4. 复杂性分析
5. 典型案例
6. 开放性讨论
7. 总结

基本思想

- 求解问题算法的复杂性一般都与问题规模相关，问题规模越小越容易处理。
- 分治法的基本思想是，将一个难以直接解决的大问题，分解为规模较小的相同子问题，直至这些子问题容易直接求解，并且可以利用这些子问题的解求出原问题的解。各个击破，分而治之。
- 分治法产生的子问题一般是原问题的较小模式，这就为使用递归技术提供了方便。递归是分治法中最常用的技术。

基本思想


原始问题的解
原始问题(N个输入)
子问题1
子问题2
…
子问题k
相同的类型
合并最小问题的解
子问题1
子问题2
…
子问题k
不用再分就可求解

应用的先决条件

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；
- 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。

基本步骤

- 一般来说，分治法的求解过程由以下三个阶段组成：
  - 划分：既然是分治，当然需要把规模为n的原问题划分为k个规模较小的子问题，并尽量使这k个子问题的规模大致相同。
  - 求解子问题：各子问题的解法与原问题的解法通常是相同的，可以用递归的方法求解各个子问题，有时递归处理也可以用循环来实现。
  - 合并：把各个子问题的解合并起来，合并的代价因情况不同有很大差异，分治算法的有效性很大程度上依赖于合并的实现。

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。

复杂性分析

- 分治法的复杂性分析依据—递推方程
- 两类递推方程：

$$T(n)=\begin{cases} O(1) & n=1 \\ \sum_{i=1}^{k} a_i T(n-i)+f(n) & n>1 \end{cases}$$

$$T(n)=\begin{cases} O(1) & n=1 \\ aT(\frac{n}{b})+f(n) & n>1 \end{cases}$$

- 求解方法：
  - 迭代法，递归树

复杂性分析

- 典型的递推方程

$$T(n)=\begin{cases} O(1) & n=1 \\ aT(\frac{n}{b})+f(n) & n>1 \end{cases}$$

- 迭代法可求得

$$T(n)=n^{\log_b a}+\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

注意：递归方程及其解只给出n等于b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b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$b^i \le n < b^{i+1}$时，$T(b^i) \le T(n) < T(b^{i+1})$。

典型案例

例

- 排列问题

问题：R是由n个元素构成的序列集合，R={r_1, r_2, … ,r_n}，求R的全排列perm(R)。

1、理解问题
2、选择策略
3、算法设计
4、正确性证明
5、算法分析
6、程序设计

典型案例

- 理解问题

(1) 若R中只有1个元素{r}，则perm(R)=(r)

(2) 若R中只有2个元素{r_1, r_2}，则

perm(R)=(r_1)perm(R_1)∪(r_2)perm(R_2)

其中，R_i=R-{r_i}

(3) 若R中有3个元素{ r_1, r_2, r_3}，则

perm(R)=(r_1)perm(R_1)∪(r_2)perm(R_2)∪(r_3)perm(R_3)

$$perm(R)=\begin{cases}(r) & n=1\\ \cup(r_i)\,perm(R_i) & n>1\end{cases}$$

典型案例

- 算法设计

思想：分治法

依次将待排列的数组的后n-1个元素与第一个元素交换,则每次递归处理的都是后n-1个元素的全排列。 当数组元素仅有一个时为此递归算法的出口。

接口： perm(Type list[], int k, int m)

典型案例

- 算法设计—伪代码

```
算法 perm(Type list[], int k, int m)
//生成列表list的全排列
//输入：一个全排列元素列表list[0..n-1]
//输出：list的全排列集合
if k == m
  for i←0 to m do
    输出list[i]
else
  for i←k to m do
    swap list[k] and list[i]
    perm(list, k+1, m)
    swap list[k] and list[i]
```

典型案例

- 算法分析

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

- 程序设计

实践

  - 编码
  - 测试

典型案例

例

- 整数划分问题

问题：将给定正整数n表示成一系列正整数之和 $n=n_1+n_2+\ldots+n_k$，其中 $n_1\geq n_2\geq\ldots\geq n_k\geq 1$，$k\geq 1$。
求正整数n的不同划分个数p(n)。

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;

典型案例

- 算法设计
  - 思想—分治?

讨论与理解

有些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。而有些问题递归关系却不明显。
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。

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

典型案例

- 算法分析
  - 思考：是否有重复计算问题？如果有的话如何解决？

扩展思考

- 程序设计

```
int q(int n, int m) {
    if ((n < 1)||(m < 1)) return 0;
    if ((n == 1)||(m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n==m) return q(n, m-1) + 1;
    return q(n, m-1) + q(n-m, m);
}
```

- 问题应用与扩展：苹果装盘问题

扩展思考与讨论

典型案例

例

- 二分搜索

问题：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。

- ✓ 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。

给定顺序序列：3，5，7，8，9，12，15
搜索9

典型案例

- 算法设计-算法分析-程序设计

实践

递归实现

非递归实现

编码

测试

$T(n)=T(n/2)+O(1)$
$T(n)= O(\log n)$

问题：如果要查找下标最小的x，如何做？

对于依赖键值比较操作的查找算法，二分搜索是一种最优的查找算法。还有更优的？插值查找，散列

典型案例

例

- 大整数乘法

问题：设X和Y都是n位二进制数，设计一个有效算法计算它们的乘积XY。

直接加运算：$O(n^2)$，能够做的更好？

分治法：

X = AB　Y = CD

$X = A\,2^{n/2} + B \quad Y = C\,2^{n/2} + D$

$XY = AC\,2^{n} + (AD+BC)\,2^{n/2} + BD$

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

$T(n)=O(n^{\log 4})=O(n^2)$ ✘没有改进

典型案例

- 大整数乘法

实验显示，n大于600位时，分治法性能才超越传统算法。

改进思想：通过代数变换，减少乘法次数。

1. $XY = AC\ 2^n + ((A-B)(D-C)+AC+BD)\ 2^{n/2} + BD$

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

$T(n)=O(n^{\log 3}) =O(n^{1.59})$

✓较大的改进

➢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。

典型案例

例

- 矩阵乘法

$$\left.\begin{array}{l} A=[a_{ij}],\ B=[b_{ij}]. \\ C=[c_{ij}]=A\cdot B. \end{array}\right\}\ i,j=1,2,\ldots,n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij}=\sum_{k=1}^{n} a_{ik}\cdot b_{kj}$$

典型案例

- 矩阵乘法

问题归结为计算元素C[i,j]

传统方法:
依据定义来计算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 n^2个元素所需的计算时间为$O(n^3)$

典型案例

- 矩阵乘法-分治

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

问题足够小情况:
如果n=2，则2个2阶方阵的乘积可以直接计算出来，共需要8次乘法和4次加法。

分治思想：将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程C=AB重写为:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11}+A_{12}B_{21} \\ C_{12} &= A_{11}B_{12}+A_{12}B_{22} \\ C_{21} &= A_{21}B_{11}+A_{22}B_{21} \\ C_{22} &= A_{21}B_{12}+A_{22}B_{22} \end{aligned}$$

计算2个n阶方阵的乘积可以转化为计算8个n/2阶方阵的乘积和4个n/2阶方阵的加法。加法可以在$O(n^2)$时间内完成。

$T(n)=O(n^{\log_b a})=O(n^{\log 7})\approx O(n^{2.81})$

✓较大的改进

典型案例

- 矩阵乘法-Strassen乘法

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

改进思想：若想降低时间复杂度，必须想办法减少乘法次数。即减小递推函数中的a，减小1也会有改进。
问题关键在于：计算2个2阶方阵的乘积时，能否用少于8次的乘积运算。

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

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

Strassen的方法用了7次对n/2阶矩阵乘积的递归调用和18次n/2阶矩阵的加减运算。

典型案例

- 矩阵乘法-是否能进一步降低时间复杂度?

能否找到一种计算2阶方阵乘法的算法，使得乘法的计算次数少于7次？

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 $\mathbf{O(n^{2.376})}$。

目前所知道的矩阵乘法下界仍然是$\Omega(n^2)$，但能否找到$O(n^2)$的算法？

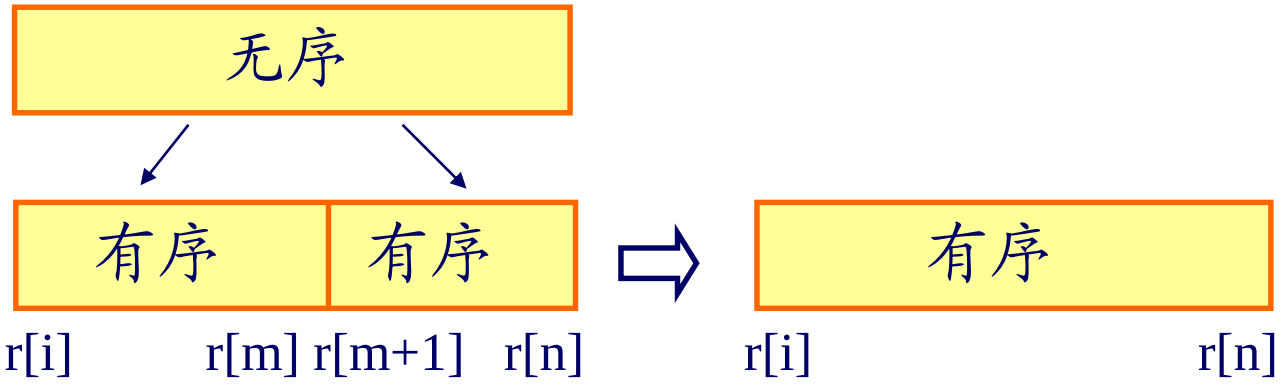
典型案例

例

- 合并排序

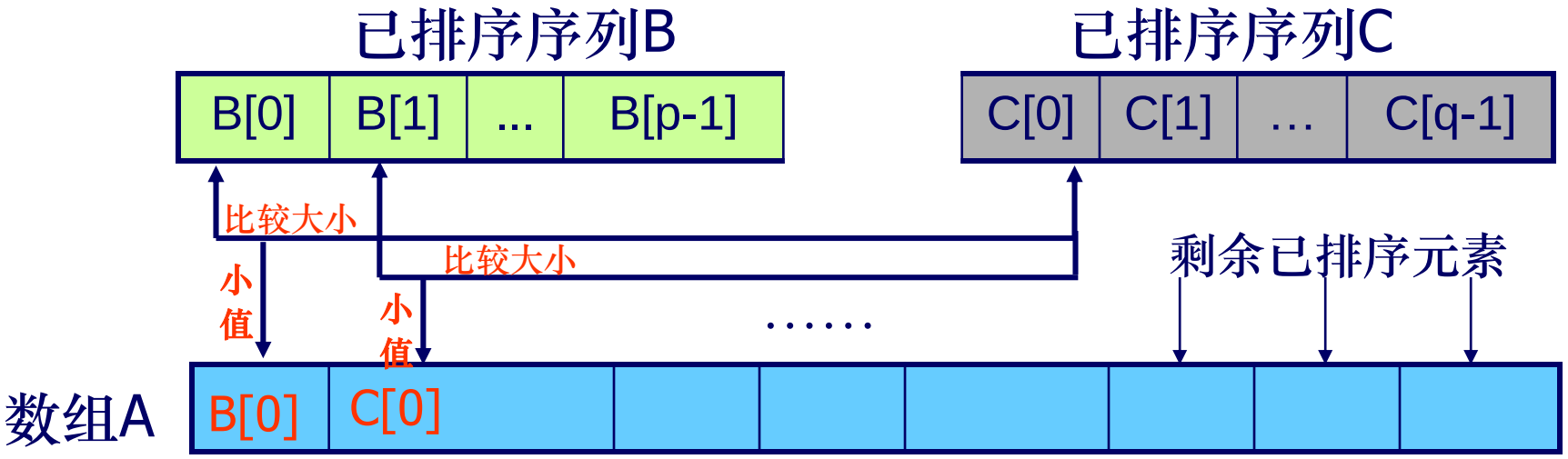
问题：给定一个可排序的n个元素序列（数字、字符或字符串），对它们按照非降序方式重新排列。

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排好序的集合。



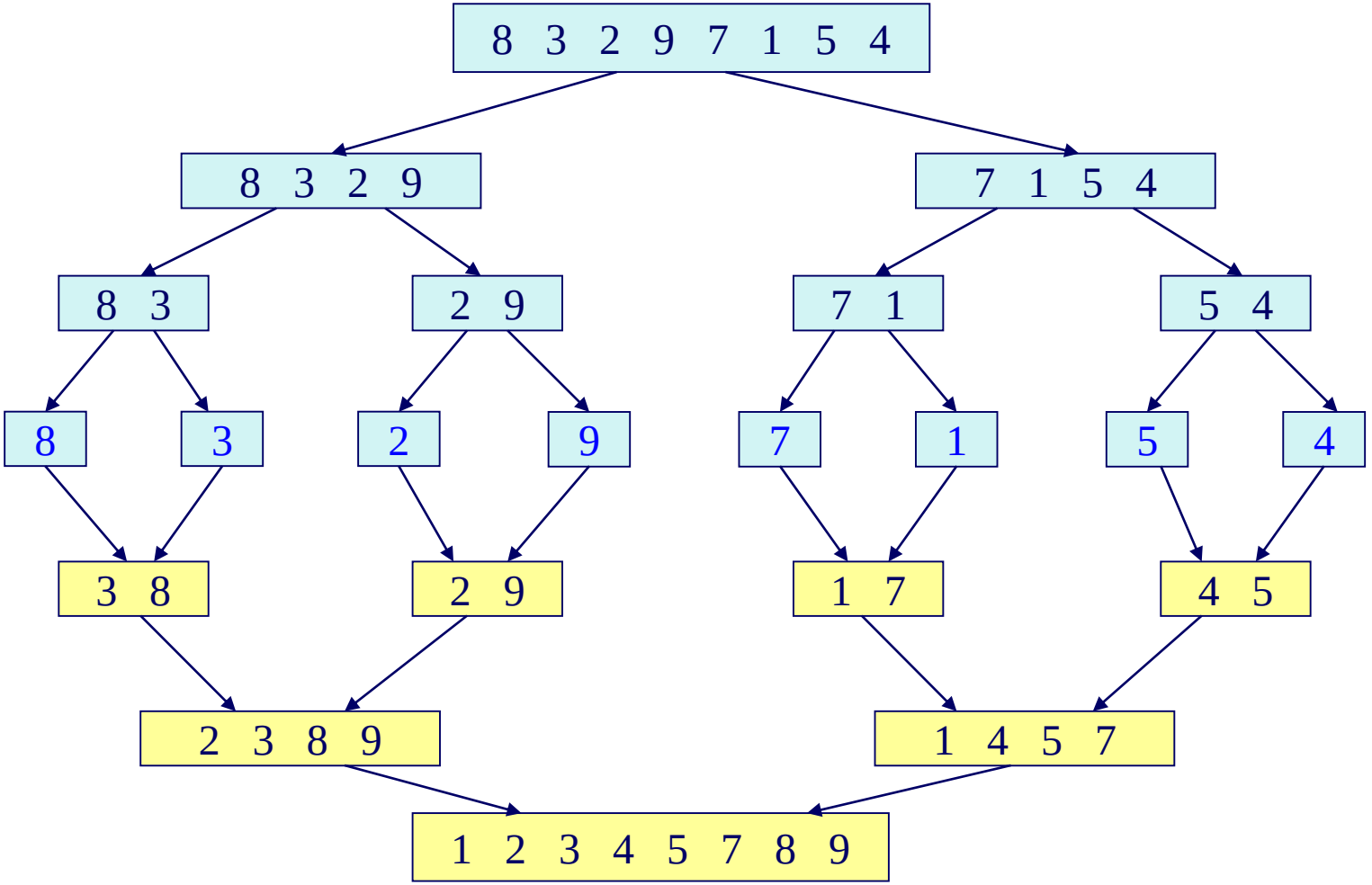
典型案例

- 合并排序-合并函数基本思想



典型案例

- 合并排序-小规模案例分析



典型案例

- 合并排序-算法设计-算法分析-程序设计

实践

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

T(n)=O(nlogn) 渐进意义下的最优算法
辅助空间O(n)

编码

测试

典型案例

- 合并排序—讨论
  - 合并排序在最坏情况下的键值比较次数十分接近于任何基于比较的排序算法在理论上能够达到的最少次数
  - 合并排序需要辅助的$O(n)$空间
  - 合并排序没能充分利用序列的特征，因此，最好情况下。。。
  - 进一步优化：自然合并

典型案例

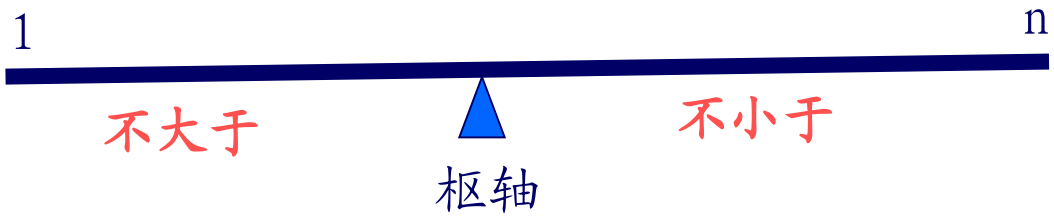
例

- 快速排序

问题：给定一个可排序的n个元素序列（数字、字符或字符串），对它们按照非降序方式重新排列。

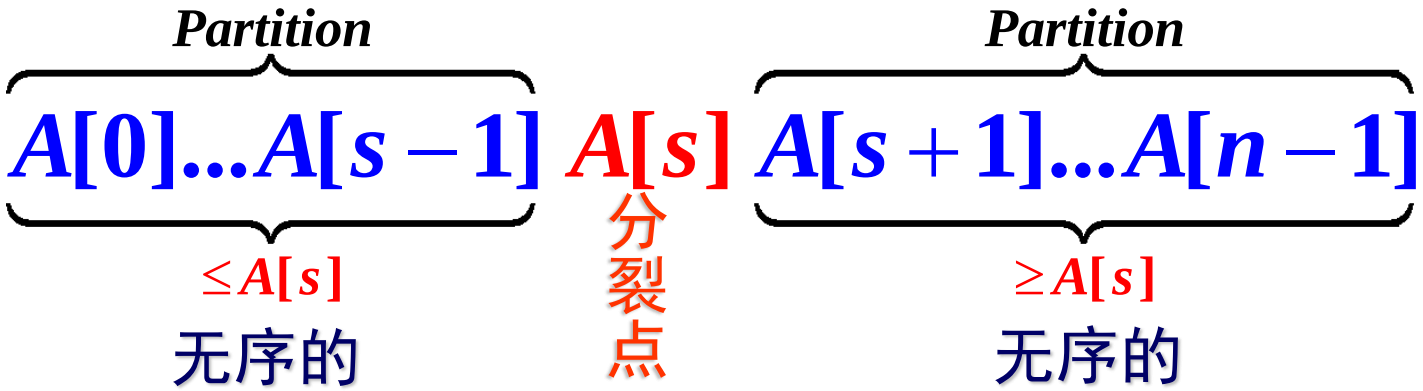
基本思想：对待排序数组不断拆分，拆分的同时不断交换元素，归位分裂点元素。记录的比较和交换是从两端向中间进行，记录每次移动的距离较大，因而总的比较和移动次数较少。

由Hoare(图灵奖获得者)1962年提出，被评为20世纪十大优秀算法之一 。



典型案例

- 快速排序—思想
  - 对给定的待排序数组不断进行拆分，这一点与和并排序思想一致
  - 合并排序按元素位置拆分（数组中间），快速排序按元素值大小拆分，得到 分区（Partition），拆分处称为 分裂点（Split position）
  - 建立分区时完成了元素的重排，拆分结束即排序结束



典型案例

- 快速排序-算法设计-算法分析-程序设计

实践

关键在于分裂点（哨兵）选取和分区函数设计

编码

测试

典型案例

- 快速排序思想延伸-kth select

问题：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。

基本思想：借鉴快速排序思想，如果将这n个元素依其线性序排列时，排在第k个位置的元素就是要找的元素。

```
template<class Type>
Type randomizedSelect(Type a[],int p,int r,int k) {
    if (p==r) return a[p];
    int i=randomizedPartition(a,p,r),
    j=i-p+1;
    if (k<=j) return randomizedSelect(a,p,i,k);
    else return randomizedSelect(a,i+1,r,k-j);
}
```

最坏时间复杂度：$O(n^2)$
平均时间复杂度：$O(n)$

最坏时间复杂度：$O(n)$?

典型案例

- 线性时间选择

如果能在线性时间内找到一个划分基准，使得按这个基准所划分出的2个子数组的长度都至少为原数组长度的ε倍(0<ε<1是某个正常数)，那么就可以**在最坏情况下**用O(n)时间完成选择任务。

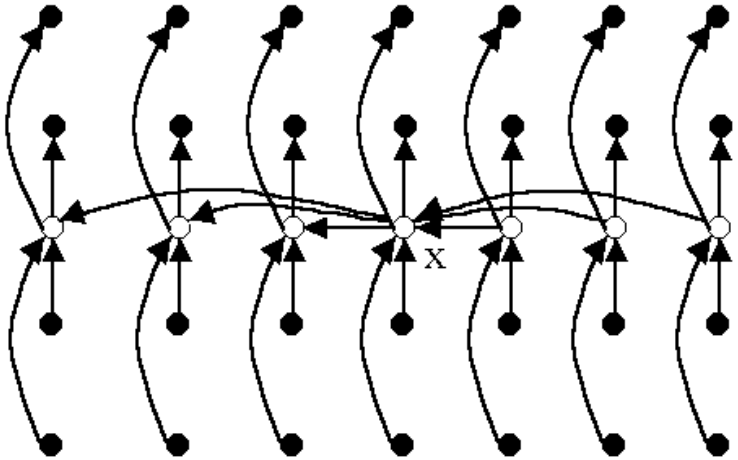
例如，若ε=9/10，算法递归调用所产生的子数组的长度至少缩短1/10。所以，在最坏情况下，算法所需的计算时间T(n)满足递归式$T(n) \leq T(9n/10)+O(n)$ 。由此可得$T(n)=O(n)$。

典型案例

例

- 线性时间选择-实例

1、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。
2、递归调用select来找出这⌈n/5⌉个元素的中位数。如果⌈n/5⌉是偶数，就找它的2个素作为划分基准。



设所有元素互不相同。在这种情况下，找出的基准x至少比3(n-5)/10个元素大，因为在每一组中有2个元素小于本组的中位数，而n/5个中位数中又有(n-5)/10个小于基准x。同理，基准x也至少比3(n-5)/10个元素小。而当n≥75时，3(n-5)/10≥n/4所以按此基准划分所得的2个子数组的长度都至少缩短1/4。

典型案例

- 线性时间选择-实例

```
Type Select(Type a[], int p, int r, int k){
   if (r-p<75) {
   //用某个简单排序算法对数组a[p:r]
      return a[p+k-1];
   }
   for ( int i = 0; i<=(r-p-4)/5; i++ )
      将a[p+5*i]至a[p+5*i+4]的第3小元素
      与a[p+i]交换位置;
    //找中位数的中位数，r-p-4即上面所说的n-5
   Type x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
   int i=Partition(a,p,r, x),
   j=i-p+1;
   if (k<=j) return Select(a,p,i,k);
   else return Select(a,i+1,r,k-j);
}
```

复杂度分析

$$T(n) \le \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \ge 75 \end{cases}$$

T(n)=**O(n)**

上述算法将每一组的大小定为5，并选取75作为是否作递归调用的分界点。这2点保证了T(n)的递归式中2个自变量之和n/5+3n/4=19n/20=εn，0<ε<1。这是使T(n)=O(n)的关键之处。当然，除了5和75之外，还有其他选择。

典型案例

- 最近点对问题

问题：给定平面上的n个点，找其中的一对点，使得在n个点组成的所有点对中，该点对之间的距离最小。

常规思想：只要将每一点与其它n-1个点的距离算出，找出其中距离最小的两点即可。
问题：效率太低，$T(n)=O(n^2)$

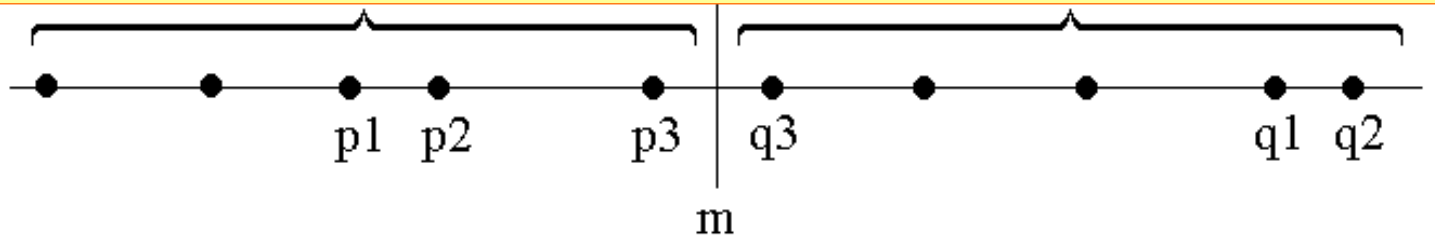
分治思想：将平面上n个点的集合S划分为两个子集S_1和S_2，每个子集约n/2个点，然后在每个子集中寻找最接近点对。
问题在于合并：如何根据子集中的最接近点对求得原集合中的最接近点对。

典型案例

- 最近点对问题

为了使问题易于理解和分析，先来考虑**一维**的情形。此时，S中的n个点退化为x轴上的n个实数 $x_1,x_2,\ldots,x_n$。最接近点对即为这n个实数中相差最小的2个实数。

假设我们用x轴上某个点m将S划分为2个子集S_1和S_2，基于平衡子问题的思想，用S中各点坐标的中位数来作分割点。递归地在S_1和S_2上找出其最接近点对$\{p_1,p_2\}$和$\{q_1,q_2\}$，并设$d=\min\{|p_1-p_2|,|q_1-q_2|\}$，S中的最接近点对或者是$\{p_1,p_2\}$，或者是$\{q_1,q_2\}$，或者是某个$\{p_3,q_3\}$，其中$p_3\in S_1$且$q_3\in S_2$。



典型案例

- 最近点对问题

➢如果S的最接近点对是{p_3,q_3}，即|p_3-q_3|<d，则p_3和q_3两者与m的距离不超过d，即**p_3∈(m-d,m]，q_3∈(m,m+d]**。

➢由于在S_1中，每个长度为d的半闭区间至多包含一个点（否则必有两点距离小于d），并且m是S_1和S_2的分割点，因此(m-d,m]中至多包含S中的一个点。因此，如果(m-d,m]中有S中的点，则此点就是S_1中最大点。

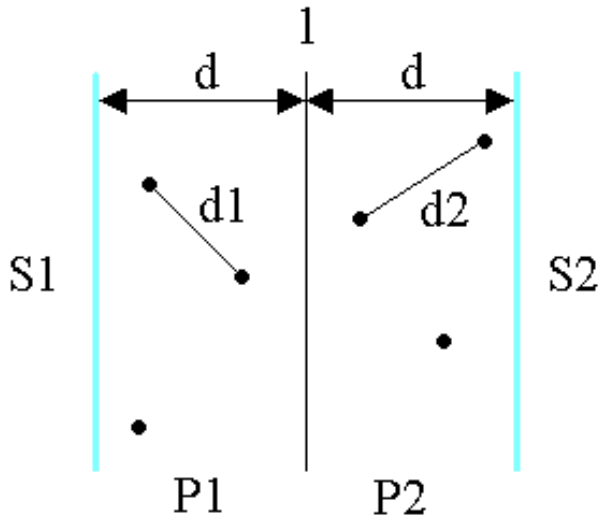
➢用线性时间就能找到区间(m-d,m]和(m,m+d]中所有点，即p_3和q_3。从而用线性时间就可以将S_1的解和S_2的解合并成为S的解。

➢二维情况？

典型案例

- 最近点对问题

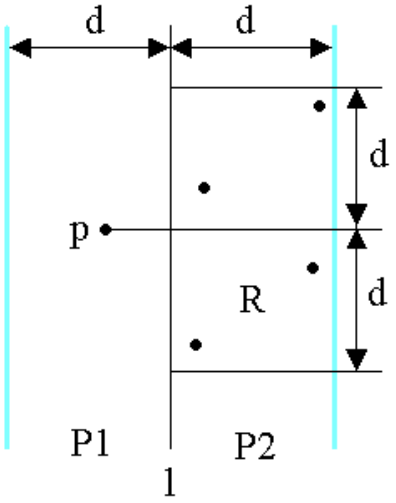
选取一垂直线**l:x=m**来作为分割直线。其中m为S中各点x坐标的中位数。由此将S分割为S_1和S_2。
递归地在S_1和S_2上找出其最小距离d_1和d_2，并设$d=\min\{d_1,d_2\}$，S中的最接近点对或者是d，或者是某个{p,q}，其中$p\in P_1$且$q\in P_2$。
能否在线性时间内找到p,q?



典型案例

- 最近点对问题

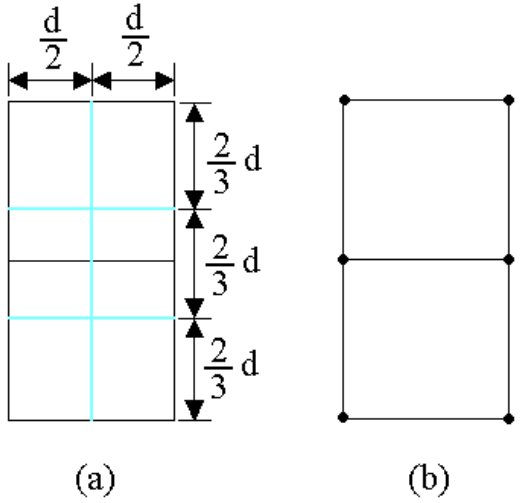
考虑P_1中任意一点p，它若与P_2中的点q构成最接近点对的候选者，则必有distance(p，q)＜d。**满足这个条件的P_2中的点一定落在一个d × 2d的矩形R中.**



典型案例

- 最近点对问题

由d的意义可知，P_2中任何2个S中的点的距离都不小于d。
由此可以推出**矩形R中最多只有6个S中的点**。
因此，在分治法的合并步骤中最多只需要检查6 × n/2=3n个候选者。



证明:将矩形R的长为2d的边3等分，将它的长为d的边2等分，由此导出6个(d/2) × (2d/3)的矩形。若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个(d/2) × (2d/3)的小矩形中有2个以上S中的点。设u，v是位于同一小矩形中的2个点，则

$$(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2=\frac{25}{36}d^2$$

distance(u,v)<d。这与d的意义相矛盾。

典型案例

- 最近点对问题

➢为了确切地知道要检查哪6个点，可以将p和P2中所有S2的点投影到垂直线l上。由于能与p点一起构成最接近点对候选者的S2中点一定在矩形R中，所以它们在直线l上的投影点距p在l上投影点的距离小于d。由上面的分析可知，这种投影点最多只有6个。

➢因此，若将P1和P2中所有S中点按其y坐标排好序，则对P1中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对P1中每一点最多只要检查P2中排好序的相继6个点。

典型案例

- 最近点对问题

bool **Cpai**

n=|S|

if (n

1、m=S

构造

//S1=

//S2={p∈S|x(p)>m}

2、**Cpai**

Cpa

3、dm=

4、设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合;

m之内

;

点列;

检查Y

多6个)

可以完成合并;

动时，Y

内移动;

点对间

复杂度分析

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

$T(n)=O(n\log^2 n)$

复杂度分析

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

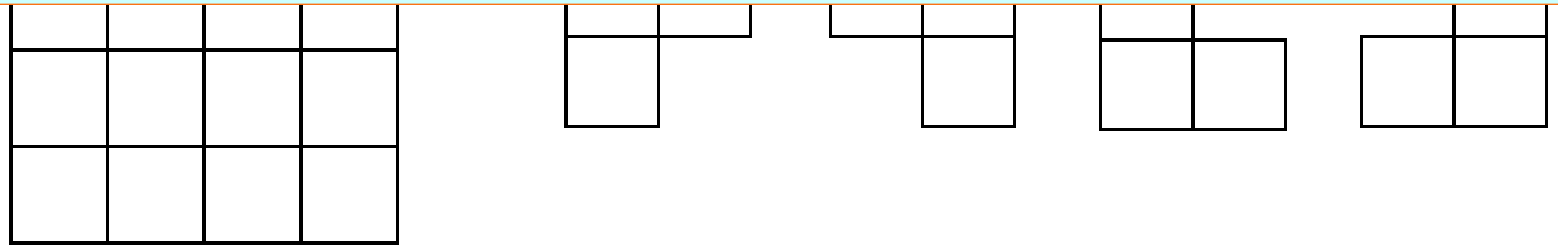
$T(n)=O(n\log n)$

典型案例

- 棋盘覆盖问题

在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其它方格不同，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。

在任何一个$2^k \times 2^k$的棋盘覆盖中，用到的L型骨牌个数恰为$(4^k-1)/3$



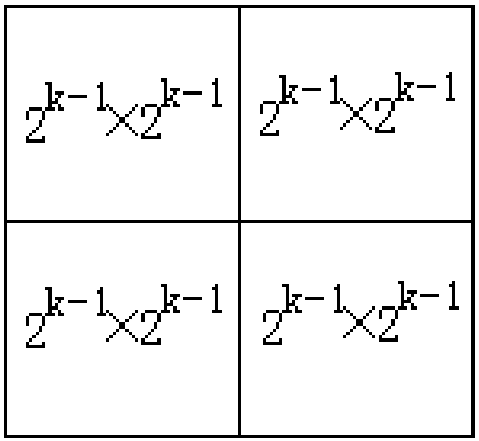
典型案例

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

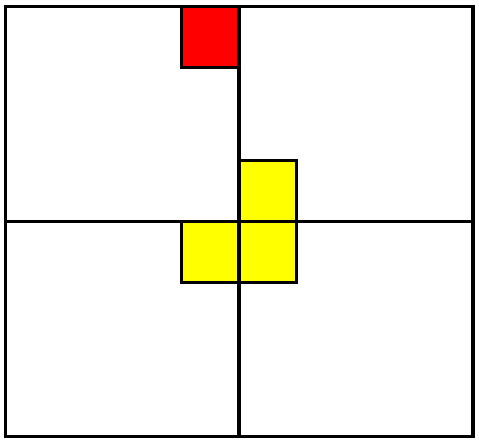
T(n)=O(4^k) 渐进意义下的最优算法

- 棋盘覆盖问题

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。



(a)


(b)

开放性讨论

- 应该把问题分解为多少个子问题合适呢?
- 分治法一定可以提高解决问题的效率么?
  - 计算n个数字$a_0, a_1, \cdots, a_{n-1}$的和。求n个元素中的最大值。计算n的阶乘
  - 计算a的n次幂
- 分治法降低复杂性的本质
- 进一步提高效率措施
  - 代数变换，减少子问题个数
  - 利用中间结果
  - 数据预处理

Summary

- 分治法基本思想
- 分治法求解问题适用条件
- 递归技术
- 分治法复杂性分析方法
- 典型分治算法