第四章 分治法

1 分治法的设计思想

2 排序问题中的分治法

3 组合问题中的分治法

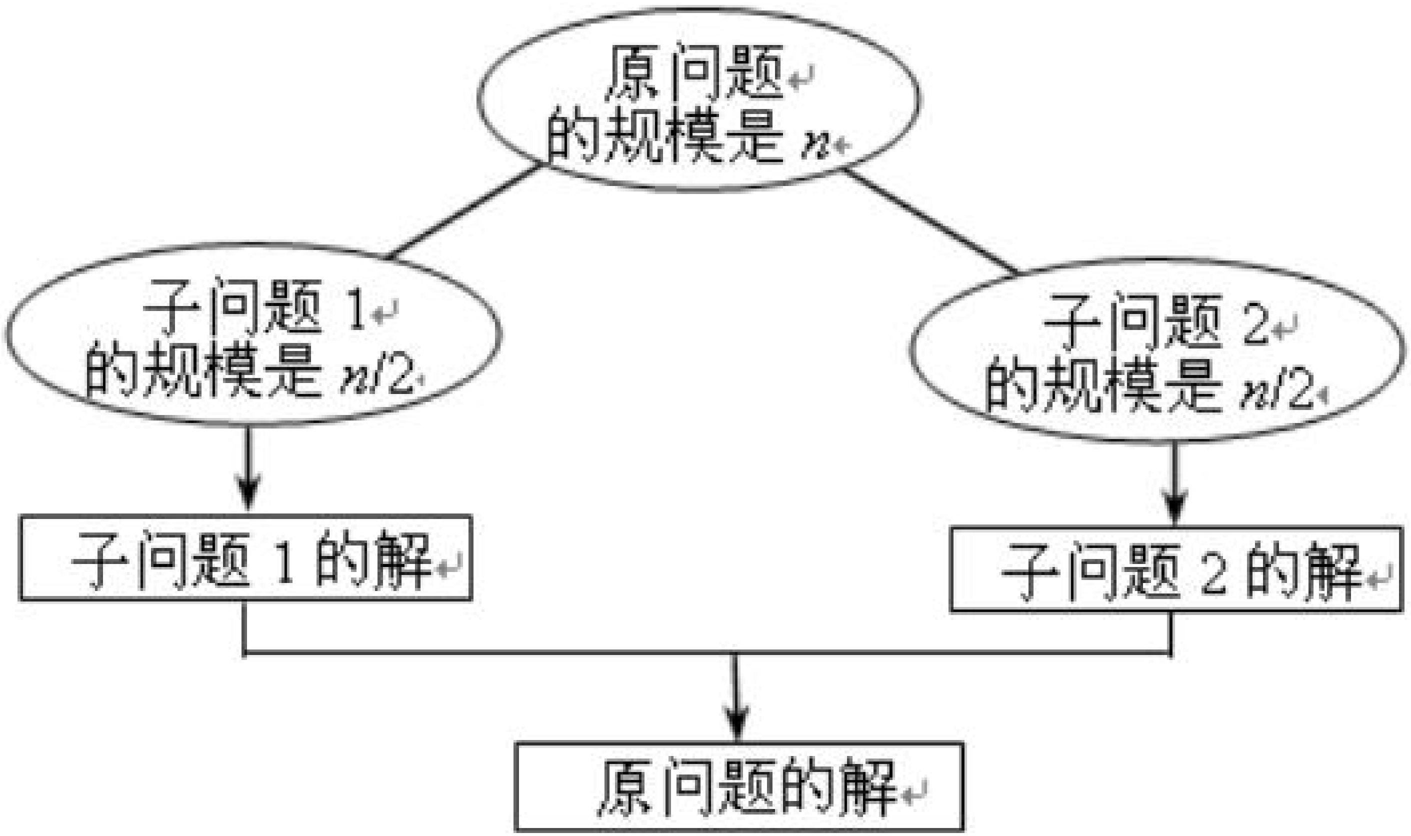
4 几何问题中的分治法

5 小结

1 分治法的设计思想

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。

1 分治法的设计思想



分治法的求解步骤

（1）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。

（2）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。

（3）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。

分治法的求解步骤

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。

分治法的求解步骤

DivideConquer(P)

{

  if (P的规模足够小) 直接求解P;

  else 分解为k个子问题P_1, P_2, …, P_k;

  for (i=1; i<=k; i++)

    y_i=DivideConquer(P_i);

  return Merge(y_1, …, y_k);

}

分治法的时间复杂度

子问题的输入规模大致相等且一分为二，则分治法的计算时间可表示为：

$$T(n)=\begin{cases} g(n) & n\text{足够小} \\ 2T(n/2)+f(n) & \end{cases}$$

说明：1. $T(n)$式输入规模为 n 的分治法的计算时间；
2. $g(n)$是对足够小的 n 直接求解的时间；
3. $f(n)$是Merge的计算时间。

分治法的适用条件

1. 问题的规模缩小到一定的程度就可以容易地解决；
2. 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；
3. 问题分解出的子问题的解可以合并为该问题的解；
4. 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。

因为问题的计算复杂性一般是随着问题规模的增加而增加，因此大部分问题满足条件1的特征。

第2条特征是应用分治法的前提，反映了递归思想的应用。

分治法的适用条件

1. 问题的规模缩小到一定的程度就可以容易地解决；
2. 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；
3. 问题分解出的子问题的解可以合并为该问题的解；
4. 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。

能否利用分治法完全取决于问题是否具有第3条特征，如果具备了前两条特征，而不具备第3条特征，则可以考虑贪心算法或动态规划。

第4条特征涉及到分治法的效率，如果各子问题不独立，则分治法要做许多重复工作，重复地求解公共子问题，此时虽然也可用分治法，但一般用动态规划较好。

2 一个简单的例子——数字旋转方阵

问题描述：输出下图所示$N\times N$（$1\le N\le 10$）的数字旋转方阵。

1	20	19	18	17	16
2	21	32	31	30	15
3	22	33	36	29	14
4	23	34	35	28	13
5	24	25	26	27	12
6	7	8	9	10	11

A、B、C、D
四个区域

2 一个简单的例子—数字旋转方阵

想法：用二维数组data[N][N]表示N × N的方阵，观察方阵中数字的规律，可以从外层向里层填数。

1.设变量size表示方阵的大小，则初始时size = N，填完一层则size = size - 2；

2. 设变量begin表示每一层的起始位置，变量i和j分别表示行号和列号，则每一层初始时i = begin，j = begin。

3. 将每一层的填数过程分为A、B、C、D四个区域，则每个区域需要填写size – 1个数字，填写区域A时列号不变行号加1，填写区域B时行号不变列号加1，填写区域C时列号不变行号减1，填写区域D时行号不变列号减1。

4. 显然，递归的结束条件是size等于0或size等于1。

2 一个简单的例子——数字旋转方阵

算法4.1：数字旋转方阵Full
输入：当前层左上角要填的数字number，左上角的坐标begin，方阵的阶数size
输出：数字旋转方阵
1. 如果size等于0，则算法结束；
2. 如果size等于1，则data[begin][begin] = number，算法结束；
3. 初始化行、列下标i = begin, j = begin；
4. 重复下述操作size – 1次，填写区域A
   4.1 data[i][j] = number; number++;
   4.2 行下标i++; 列下标不变；
5. 重复下述操作size – 1次，填写区域B
   5.1 data[i][j] = number; number++;
   5.2 行下标不变；列下标j++;
6. 重复下述操作size – 1次，填写区域C
   6.1 data[i][j] = number; number++;
   6.2 行下标i--; 列下标不变；
7. 重复下述操作size – 1次，填写区域D
   7.1 data[i][j] = number; number++;
   7.2 行下标不变，列下标j--;
8. 调用函数Full在size-2阶方阵中左上角begin+1处从数字number开始填数；

排序问题中的分治法—归并排序

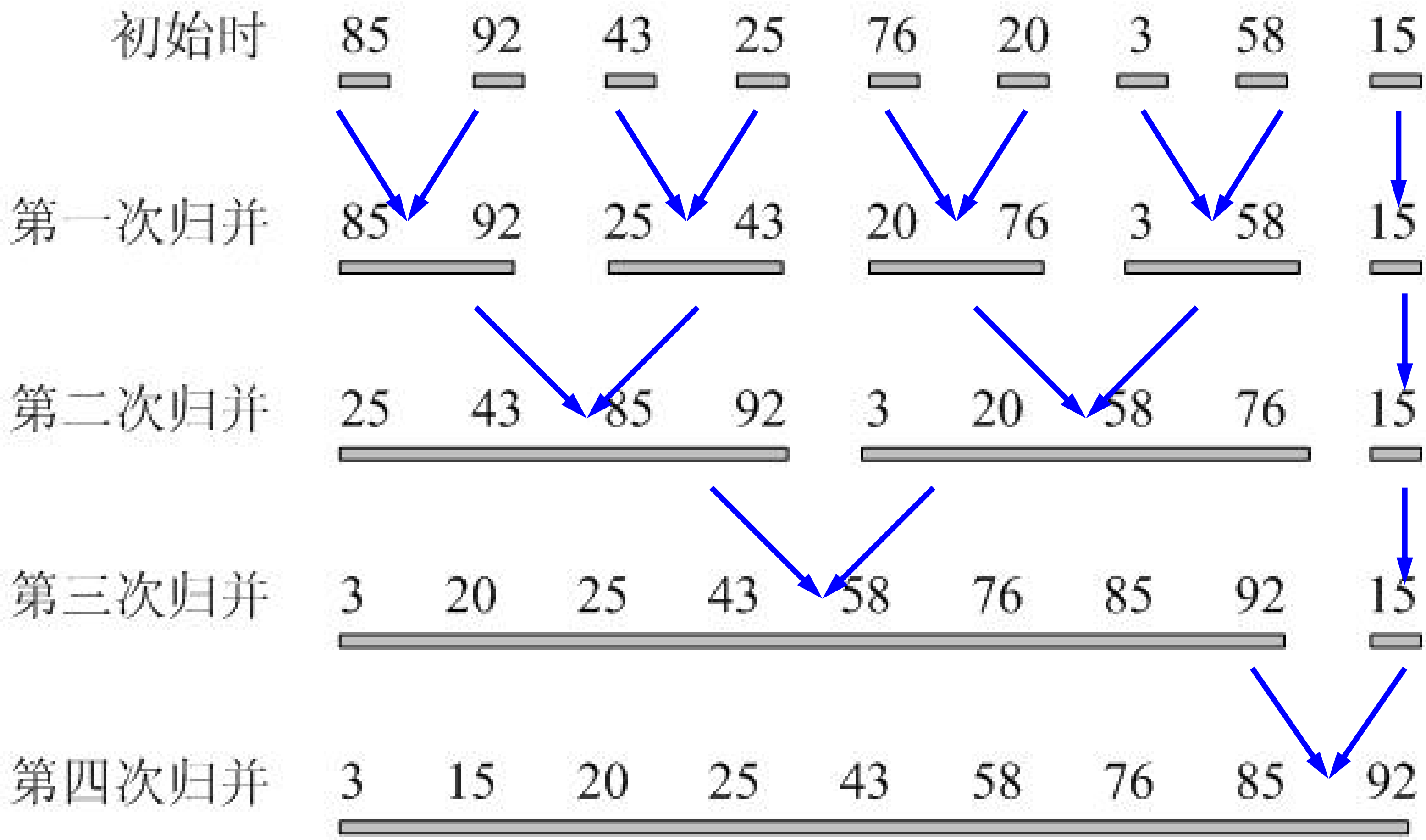
将两个有序序列合并为一个有序序列的过程称为二路归并。

二路归并排序

注意：只含一个记录的序列显然是有序序列。

排序问题中的分治法—归并排序

已知关键字序列{ 85, 92, 43, 25, 76, 20, 3, 58, 15 }，请给出二路归并排序的每一趟结果。



合并两个有序表

相邻有序表 A[*s*] ~ A[*m*]、A[*m*+1]~A[*e*]
结果得到有序表 A[*s*] ~ A[*e*]

算法:

1. 从数组A[s]~A[m]和A[m+1]~A[e]中各取一最小数;
2. 比较取出的两个数，将较小数按顺序放入数组R;
3. 从较小数对应的数组中取出下一个最小数;
4. 重复步骤2、3直到两个序列中的数据全部取走;
5. 如果A[s]~A[m]或A[m+1]~A[e]有未取走的数据，则将剩下的数全部按顺序拷贝到R中已有数据之后;
6. 将R中数据逐一拷贝回A中;

一趟归并操作

长度为*n*的序列A[ ]中每个有序序列长度为len。

一趟二路归并须考虑以下问题：

- 若*n*可以被2*len整除， 则A刚好可以合并序列为*n*/(2*len) 个长度为2*len的有序表
- 若*n*不能被2*len整除时有两种情况：

  n/(2*len)余数小于等于len， 剩下一个有序表，此时剩余元素不必再进行归并操作

  n/(2*len)余数大于len， 剩下一大一小两个有序表合并为一个有序表

初始时 85 92 43 25 76 20 3 58 15

len=1, n/2*len=9/2=4, 余数1≤len，符合情况①

n=9

第一次归并 85 92 25 43 20 76 3 58 15

len=2, n/2*len=9/4=2, 余数1≤len，符合情况①

第二次归并 25 43 85 92 3 20 58 76 15

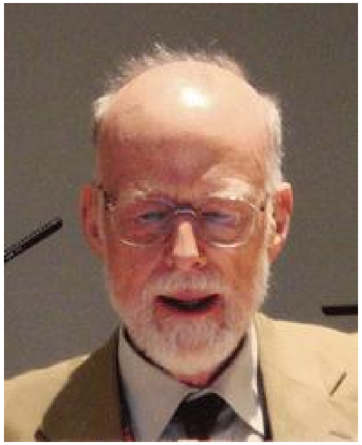
len=4, n/2*len=9/8=1, 余数1≤len，符合情况①

第三次归并 3 20 25 43 58 76 85 92 15

len=8, n/2*len=9/16=0, 余数9>len，符合情况②

第四次归并 3 15 20 25 43 58 76 85 92

排序问题中的分治法—快速排序



快速排序（Quicksort）是Hoare在26岁时给出的一个算法。

1986年，何积丰和Hoare提出了"程序分解算子"，并将规范语言与程序语言看成是同一类数学对象。

排序问题中的分治法—快速排序

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

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

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

排序问题中的分治法—快速排序

```
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);
              Mergesort(r, r1, m+1, t);
              Merge(r1, r, s, m, t);
    }
}
```

```
void Merge(int r[ ], int r1[ ], int s, int m, int t)
{    i=s; j=m+1; k=s;
     while (i<=m && j<=t)
     {    if (r[i]<=r[j])  r1[k++]=r[i++];
          else    r1[k++]=r[j++];
     }
          while (i<=m)  r1[k++]=r[i++];
          while (j<=t)    r1[k++]=r[j++];
}
```

排序问题中的分治法—快速排序

二路归并排序算法的递推式:

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

当$n=2^k$时，可得

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

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

$=4(2T(n/8)+n/4)+2n$

……

$=2^kT(1)+kn$

$=n\log_2 n$

$T(n)=O(n\log_2 n)$。

组合问题中的分治法—最大子段和问题

给定由n个整数（可能有负整数）组成的序列$(a_1, a_2, \ldots, a_n)$，最大子段和问题要求该序列形如$\sum_{k=i}^{j} a_k$的最大值（$1 \leqslant i \leqslant j \leqslant n$），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$\sum_{k=2}^{4} a_k = 20$

组合问题中的分治法—最大子段和问题

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$(a_1, a_2, \ldots, a_n)$划分成长度相同的两个子序列$(a_1, \ldots, a_{\lfloor n/2 \rfloor})$和$(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$，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 i 和j满足：

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

组合问题中的分治法—最大子段和问题

（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 $s1=\max\sum_{k=i}^{\lfloor n/2\rfloor}a_k\,(1\le i\le\lfloor n/2\rfloor)$ 和 $s2=\max\sum_{k=\lfloor n/2\rfloor+1}^{j}a_k\,(\lfloor n/2\rfloor+1\le j\le n)$，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。

组合问题中的分治法—最大子段和问题

$\underbrace{a_1 \ldots \ldots a_i \ldots a_{mid}}\ |\ \underbrace{a_{mid+1} \ldots a_j \ldots \ldots a_n}$ --------- 划分

leftsum　　rightsum -------- 递归处理

$\max\{\text{leftsum, sum, rightsum}\}$ -------- 合并解

sum -------- 不能递归处理

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

组合问题中的分治法—最大子段和问题

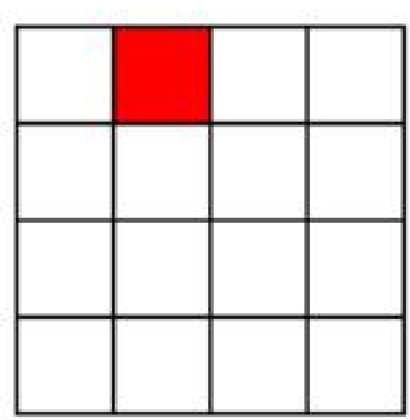
分析算法的时间性能，对应划分得到的情况①和②，需要分别递归求解，对应情况③，两个并列for循环的时间复杂性是$O(n)$，所以，存在如下递推式：

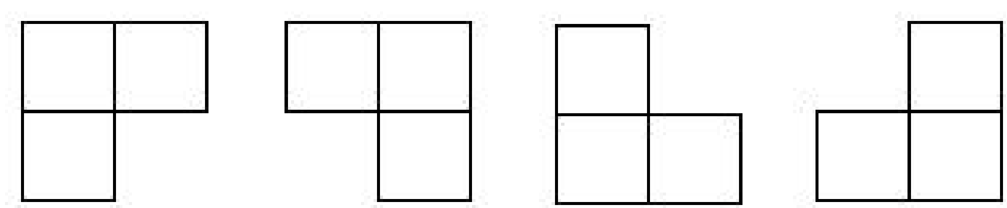
$$T(n)=\begin{cases}1 & n=1\\ 2T(n/2)+n & n>1\end{cases}$$

算法的时间复杂性为$O(n\log_2 n)$。

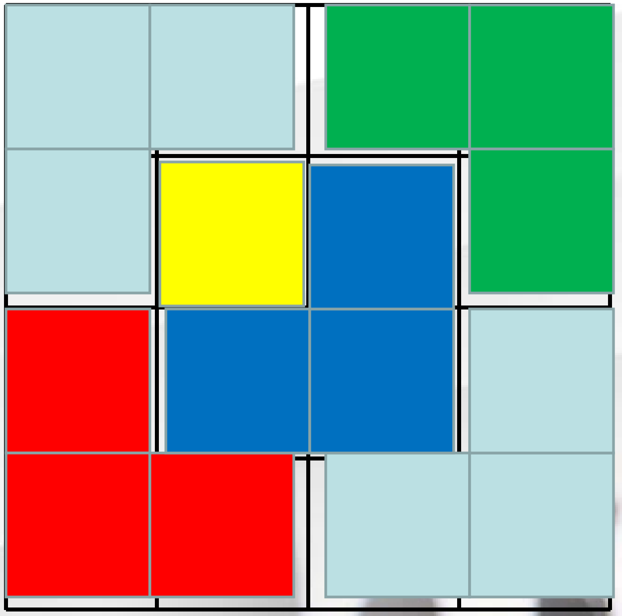
组合问题中的分治法—棋盘覆盖问题

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。





棋盘覆盖问题



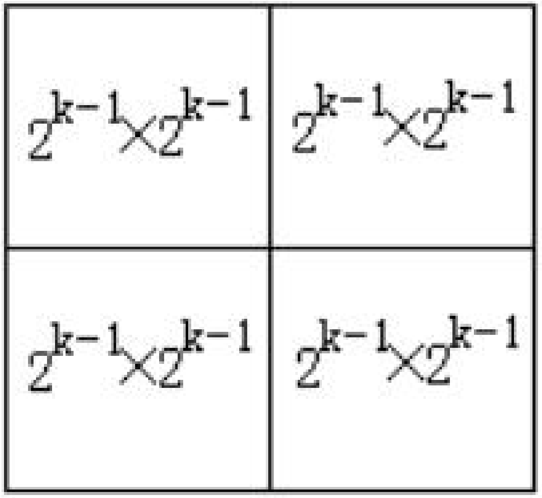
K=2时， 特殊方格在(2,2)位置上的一种棋盘，其中的一种覆盖方法

组合问题中的分治法—棋盘覆盖问题

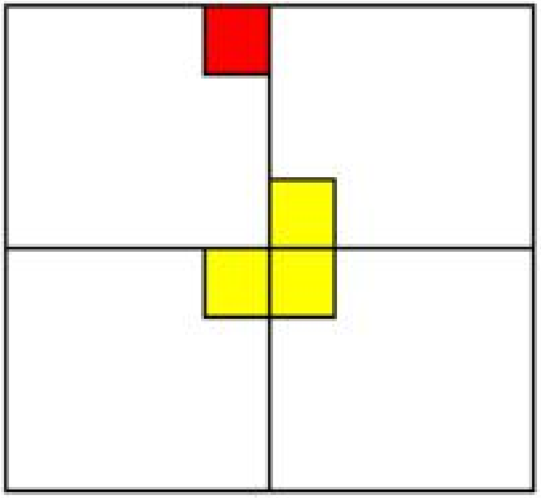
当$k>0$时，将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$ 子棋盘，特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。

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，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)

组合问题中的分治法—棋盘覆盖问题

```
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 + 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);}
   // 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
   if (dr >= tr + s && dc < tc + s)
     // 特殊方格在此棋盘中
     chessBoard(tr+s, tc, dr, dc, s);
   else {// 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);}
 }
```

组合问题中的分治法—棋盘覆盖问题

设$T(k)$是算法覆盖一个$2^k \times 2^k$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$(4^k-1)/3$。

几何问题中的分治法—最近对问题

设$p_1=(x_1,y_1)$, $p_2=(x_2,y_2)$, …, $p_n=(x_1,y_1)$，是平面上n个点构成的集合S，最近对问题就是找出集合S中距离最近的点对。

你想到些什么?

几何问题中的分治法—最近对问题

最近对问题的分治策略是:

(1)划分：将集合S分成两个子集S_1和S_2，设集合S的最近点对是p_i和p_j（$1 \le i,j \le n$），则会出现以下三种情况:

①

哪三种情况?

②

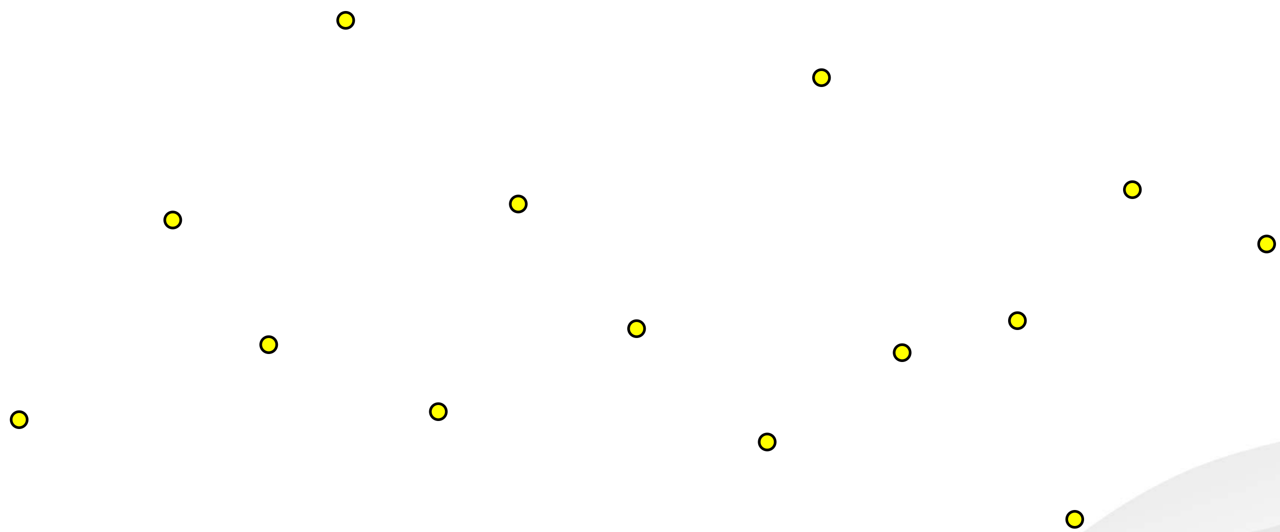
③

(2)求解子问题：对于划分阶段的情况①和②可递归求解，如果最近点对分别在集合S_1和S_2中，问题就比较复杂了。

(3)合并：比较在划分阶段三种情况下最近点对，取三者之中较小者为原问题的解。

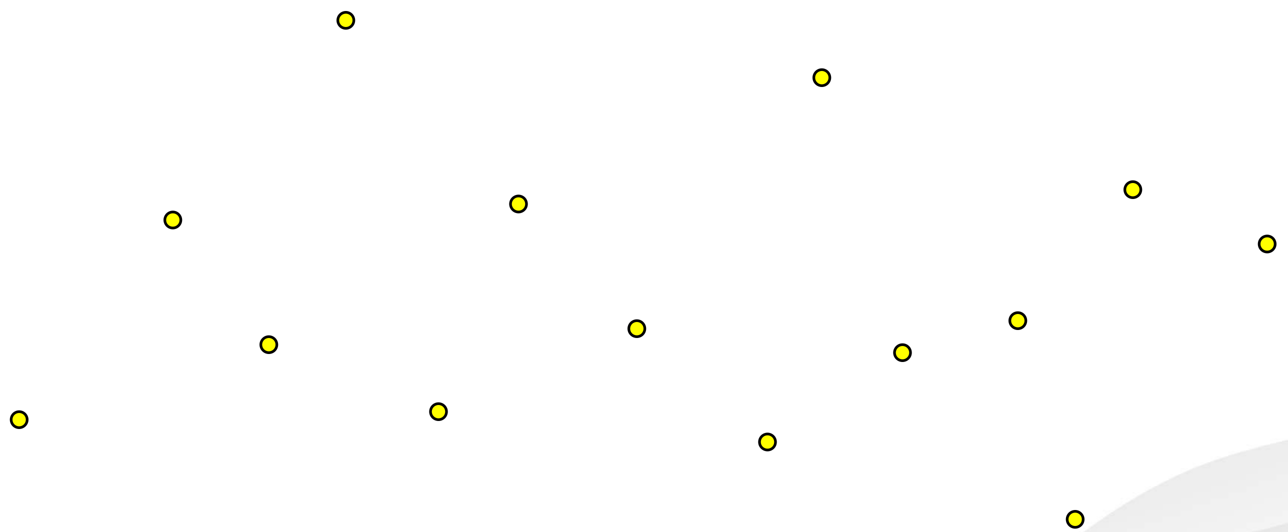
最近点对问题的分治法

给定： 一个二维平面的点集合



最近点对问题的分治法

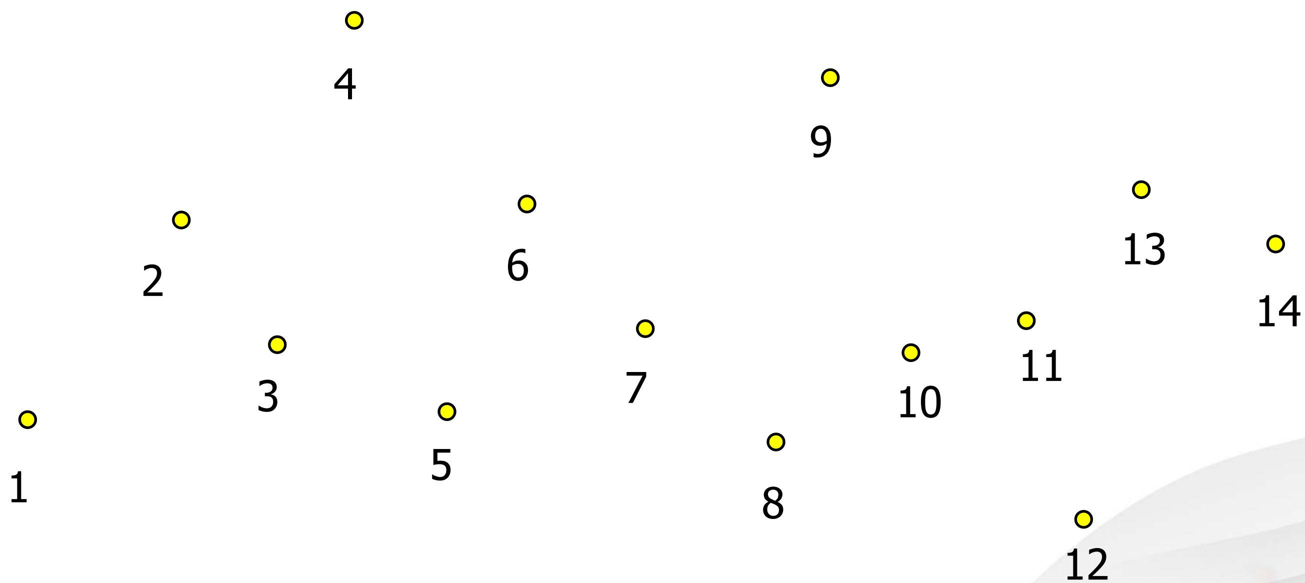
步骤一：对点集合进行一维排序



最近点对问题的分治法

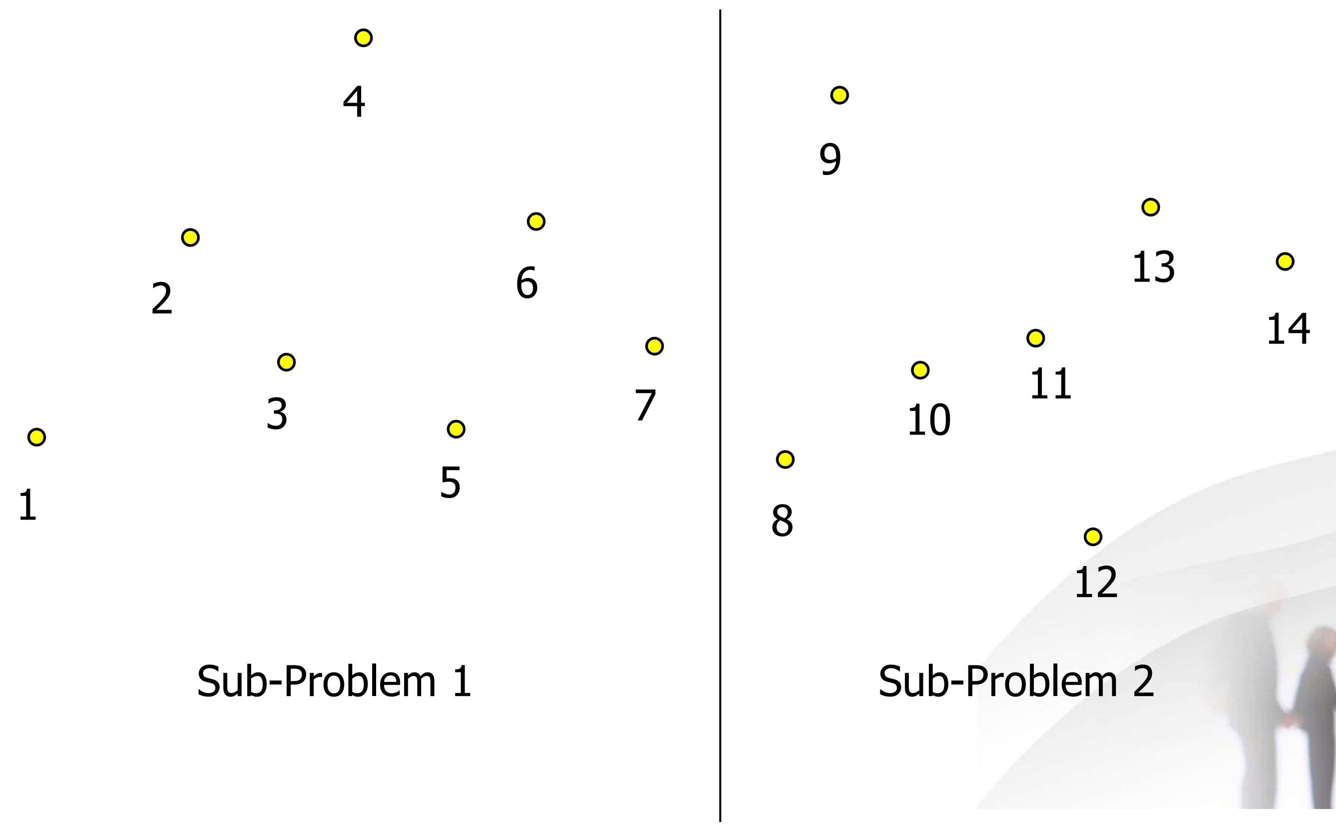
基于X-axis的排序

O(nlog n) 使用快速排序或归并排序



最近点对问题的分治法

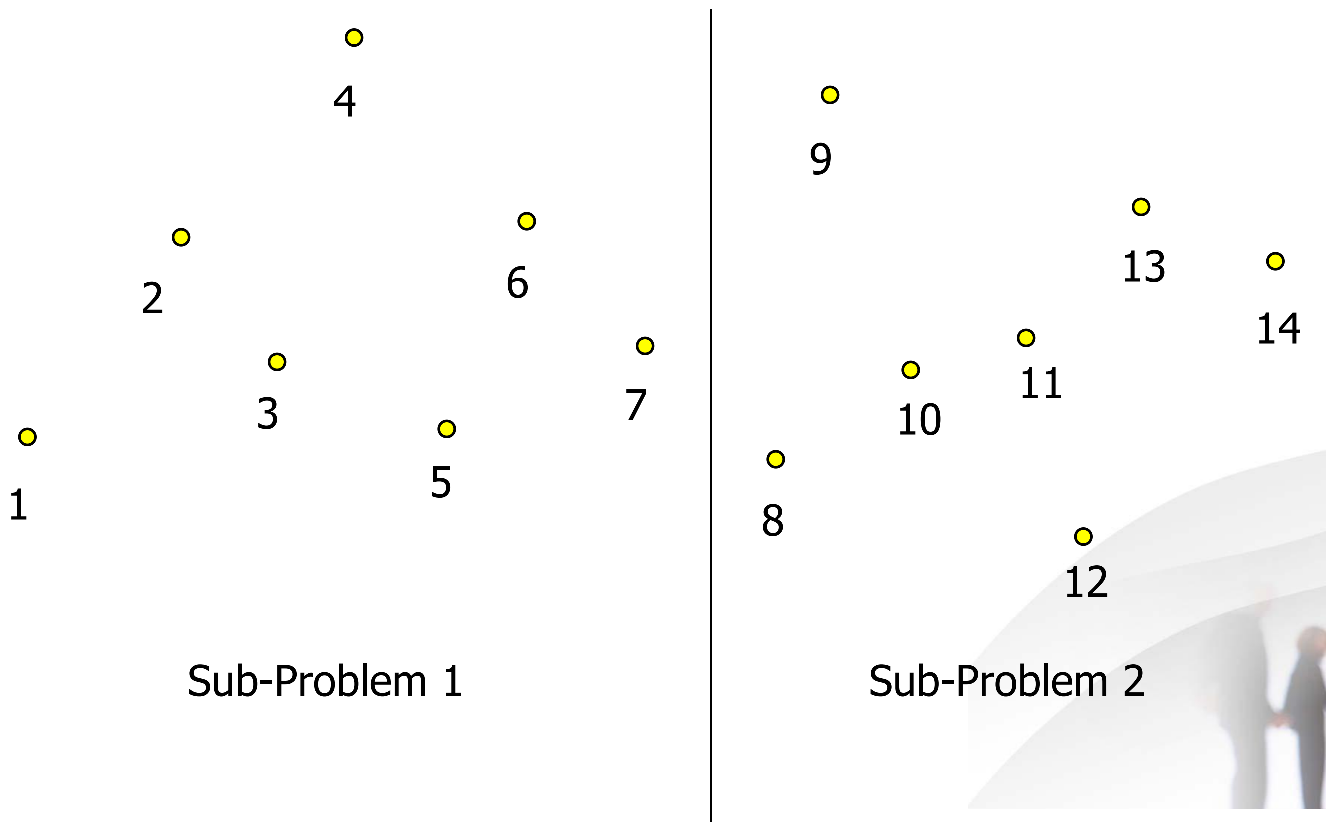
步骤二: 分割点集，即在7到8之间的中点画一条线



最近点对问题的分治法

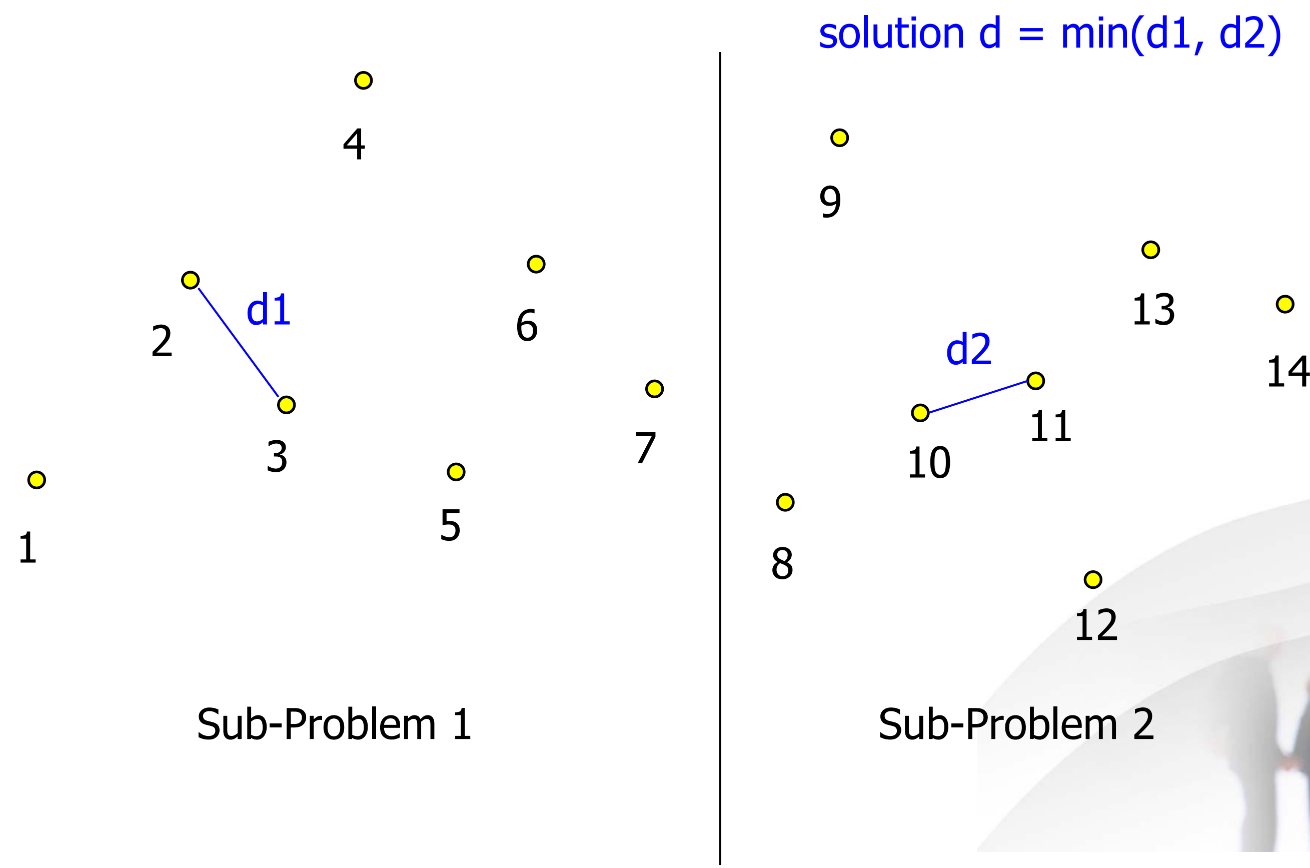
好处: 通常， 我们必须将**14**个点中的每个点与其他点进行比较。

(n-1)n/2 = 13*14/2 = **91** 次比较



最近点对问题的分治法

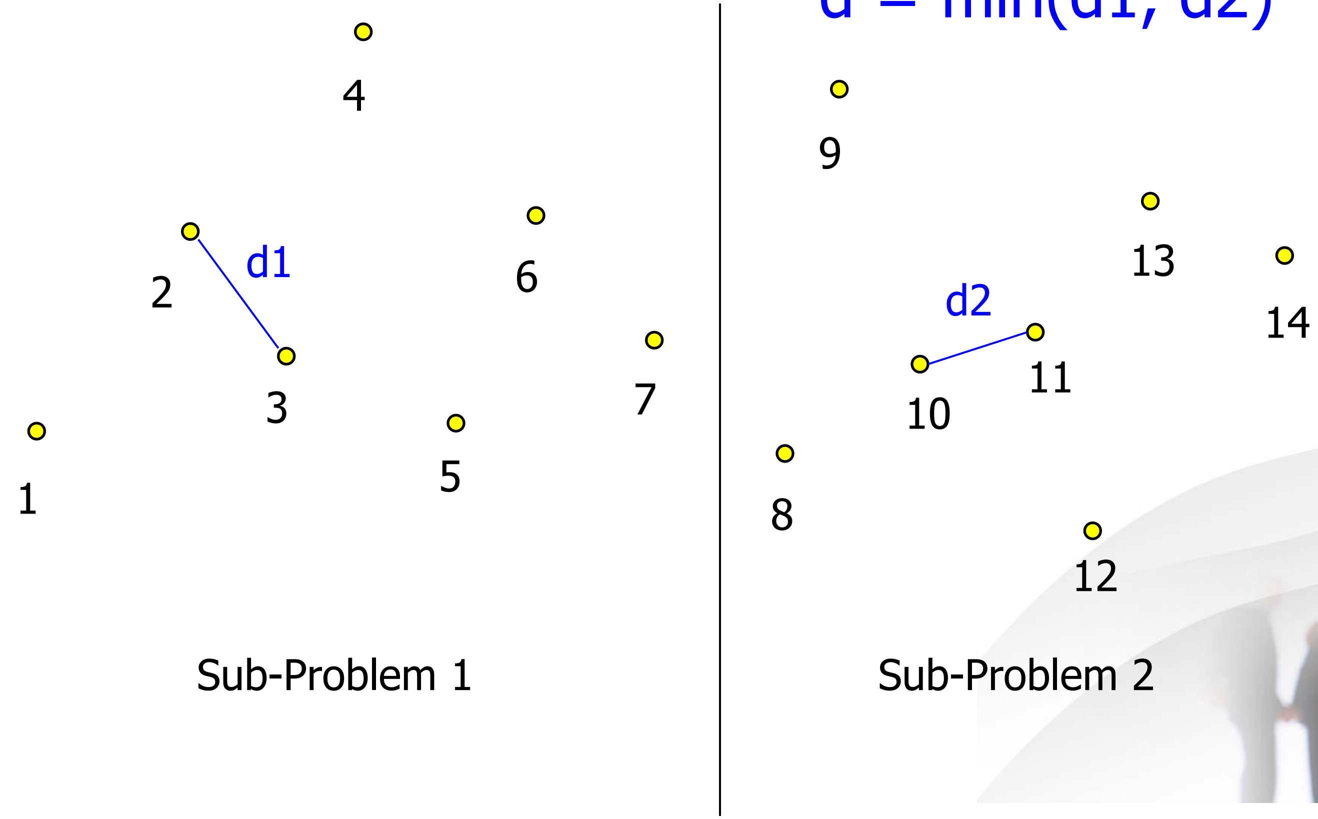
好处: 现在，我们有两个大小只有一半的子问题。因此，我们必须进行两次6 * 7/2比较，即42次比较



最近点对问题的分治法

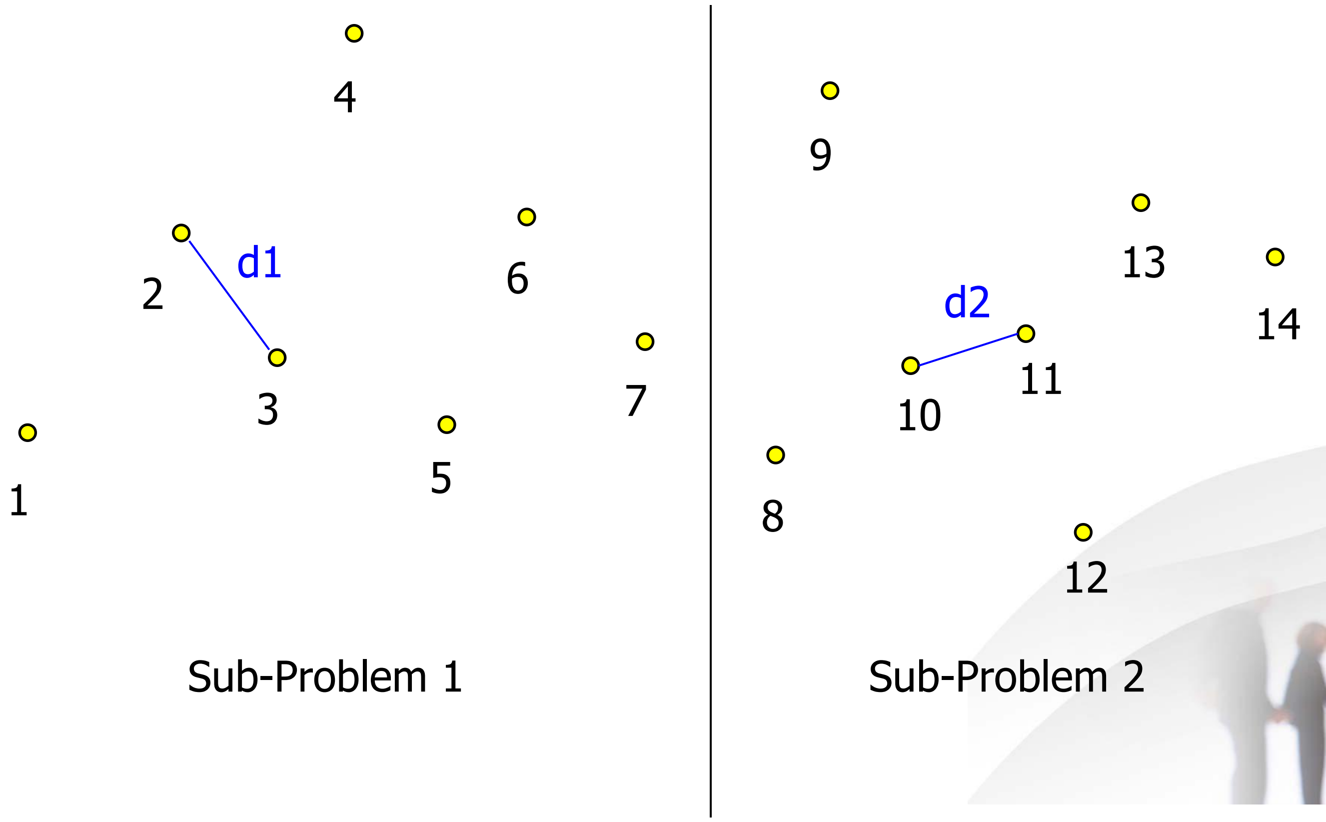
好处: 只需进行一次拆分， 我们就可以将比较次数减少一半。 显然， 如果拆分子问题， 我们将获得更大的优势。

$d = \min(d1, d2)$



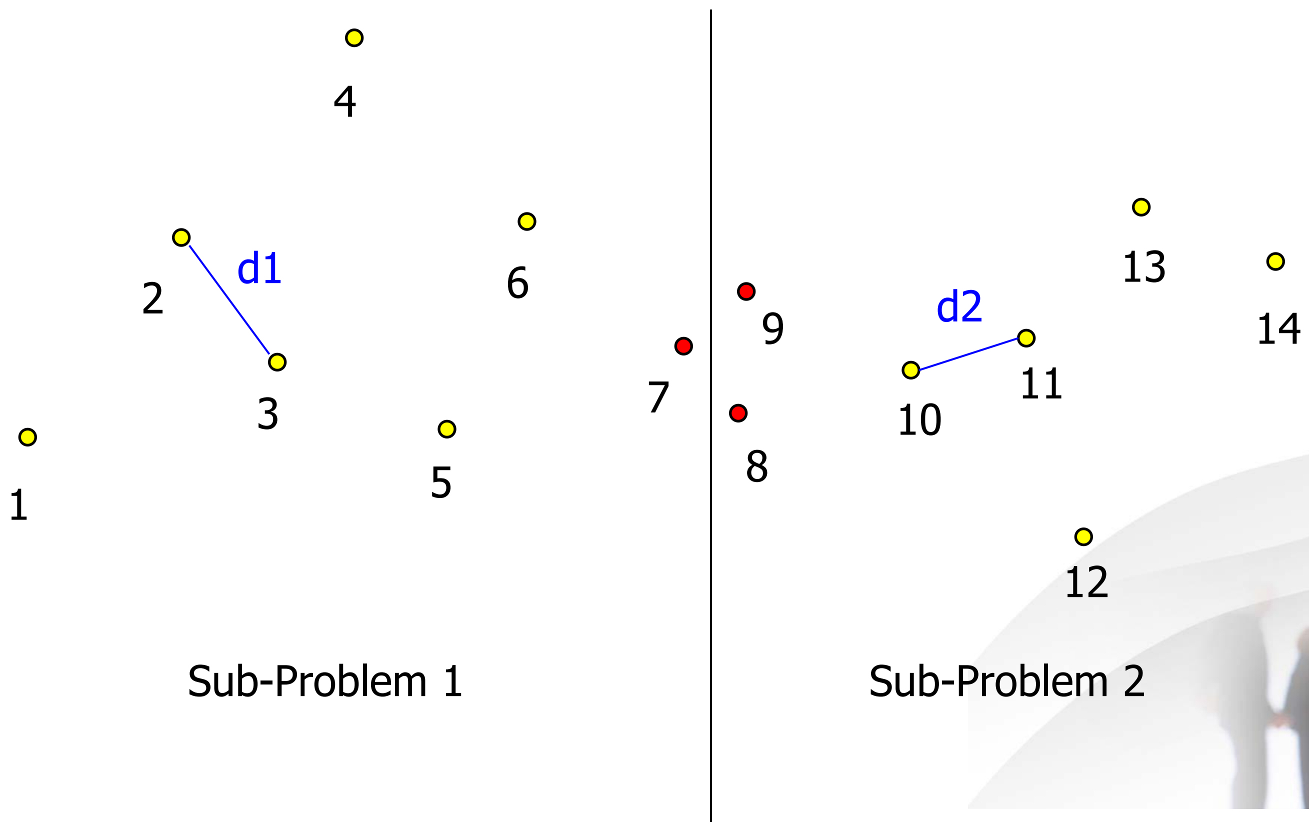
最近点对问题的分治法

问题: 但是，如果最接近的两个点分别来自不同的子问题，该怎么办?



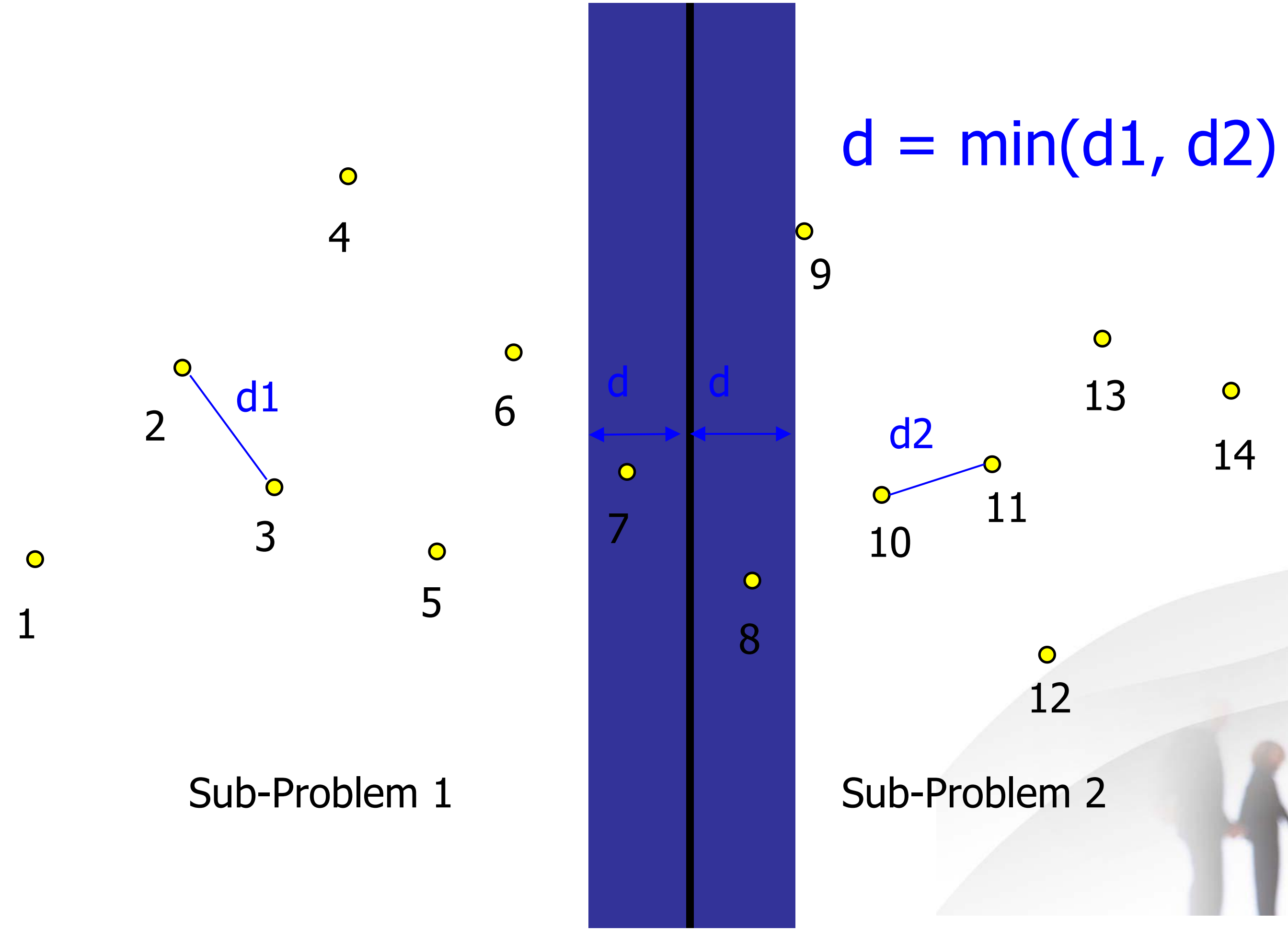
最近点对问题的分治法

这是一个示例，其中我们必须比较子问题1中的点与子问题2中的点。



最近点对问题的分治法

但是，我们只需要比较以下"条带"内的点。



小结

实际应用问题 → 构建相应模型
审题，找出题意
与结论间的数学关系

构建相应模型 → 与有关的数学问题
分析
转化

与有关的数学问题 → 数学问题的解
运用相关知识求解

数学问题的解 → 实际应用问题
翻译
作答