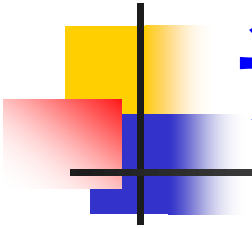
算法分析与设计

Analysis and Design of Algorithm

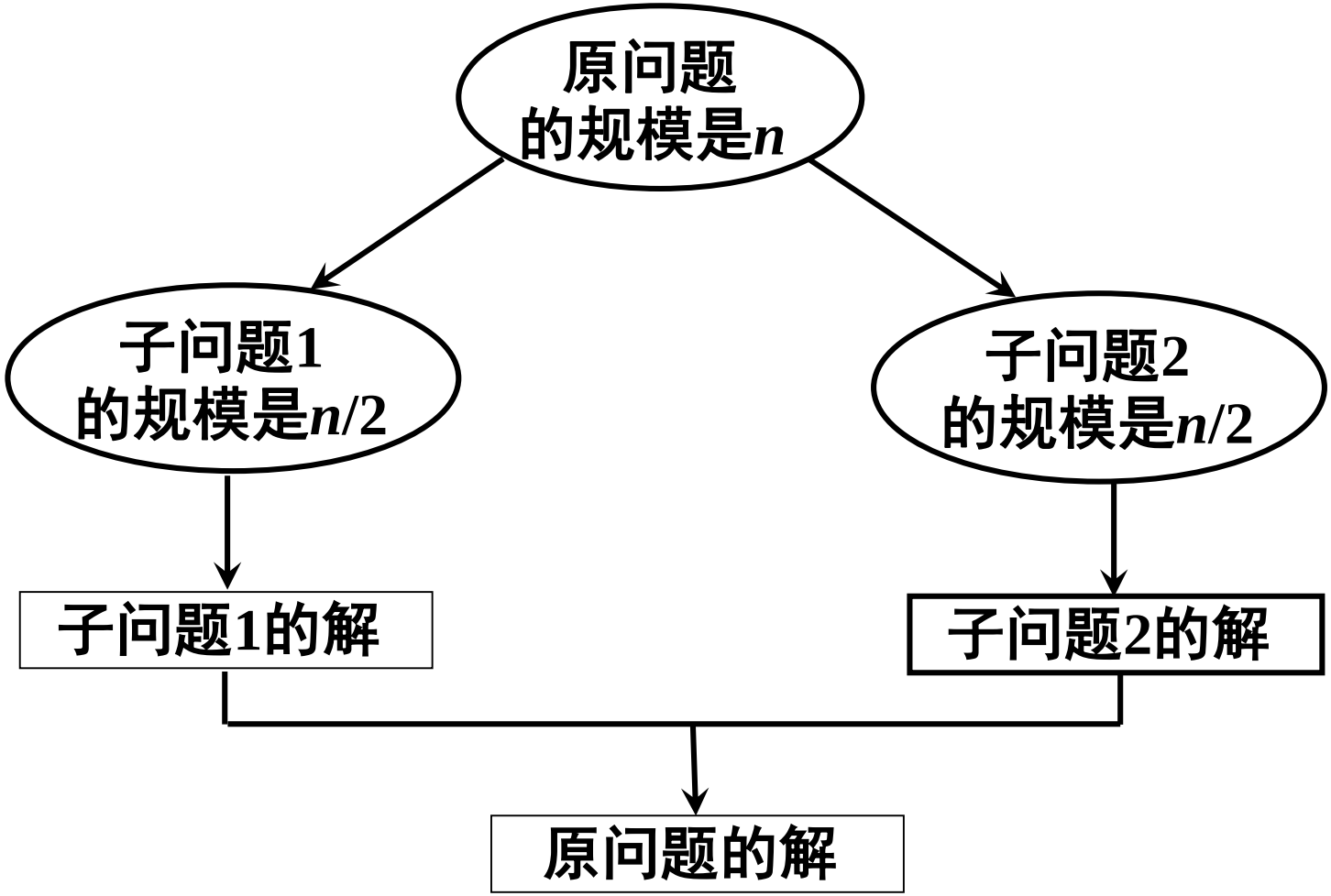
第5次课



课程回顾

- 理解分治算法的概念
- 分治法算法的时间复杂度
  - 迭代法求解递推方程
  - 差消法求
  - 递归树

分治法的典型情况



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。

二分归并排序/合并排序

二分归并排序的分治策略是：

- **划分**：将待排序序列r_1，r_2，…，r_n划分为两个长度相等的子序列r_1，…，$r_{n/2}$和$r_{n/2+1}$，…，r_n；
- **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；
- **合并**：将这两个有序子序列合并成一个有序序列。

二分归并排序/合并排序

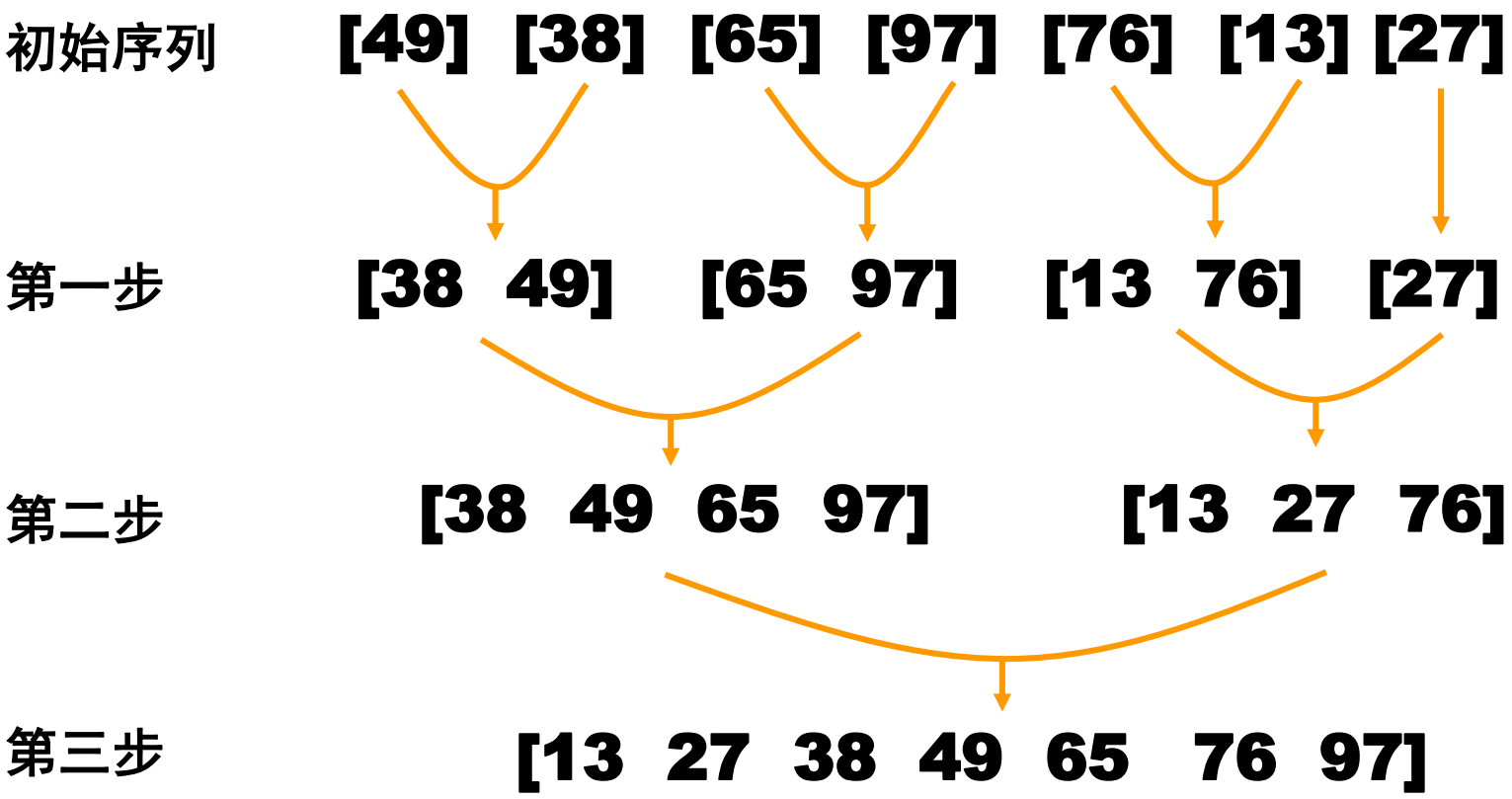
```
void mergeSort(int[] A,int[] temp, int start,int end){
    if (start<end){
        int mid = (start+end)/2;
        //把数组分解为两个子列
        mergeSort(A,temp,start,mid);
        mergeSort(A,temp,mid+1,end);
        //逐级合并两个子列
        Merge(A,temp,start,mid,end);
    }
}
```

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

$T(n)=O(n\log n)$ 渐进意义下的最优算法

二分归并排序/合并排序

改进：算法mergeSort的递归过程可以消去。



二分归并排序/合并排序复杂度

- 最坏时间复杂度： $O(n\log n)$
- 平均时间复杂度：$O(n\log n)$
- 辅助空间：$O(n)$

可以看出，不论是最坏情况还是平均情况，二分归并排序的复杂度是最好的！且是个稳定排序。

但它的一个重大缺点是，它不是一个就地操作的算法，需要O(n)个额外存储单元，当n很大的时候是一个很大的开销。

课堂练习

- 用递归树和公式法求$T(n)$
- $$T(n)=\begin{cases}1 & n=1\\ 3T(n/2)+O(n) & n>1\end{cases}$$

改进分治算法的途径1：优化子问题的划分

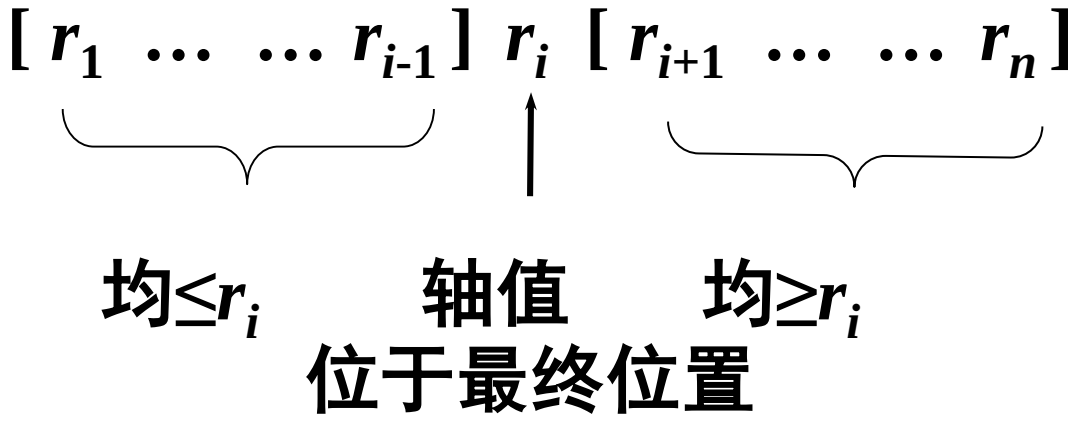
中国MOOC：3.5 快速排序

快速排序

快速排序的分治策略是：

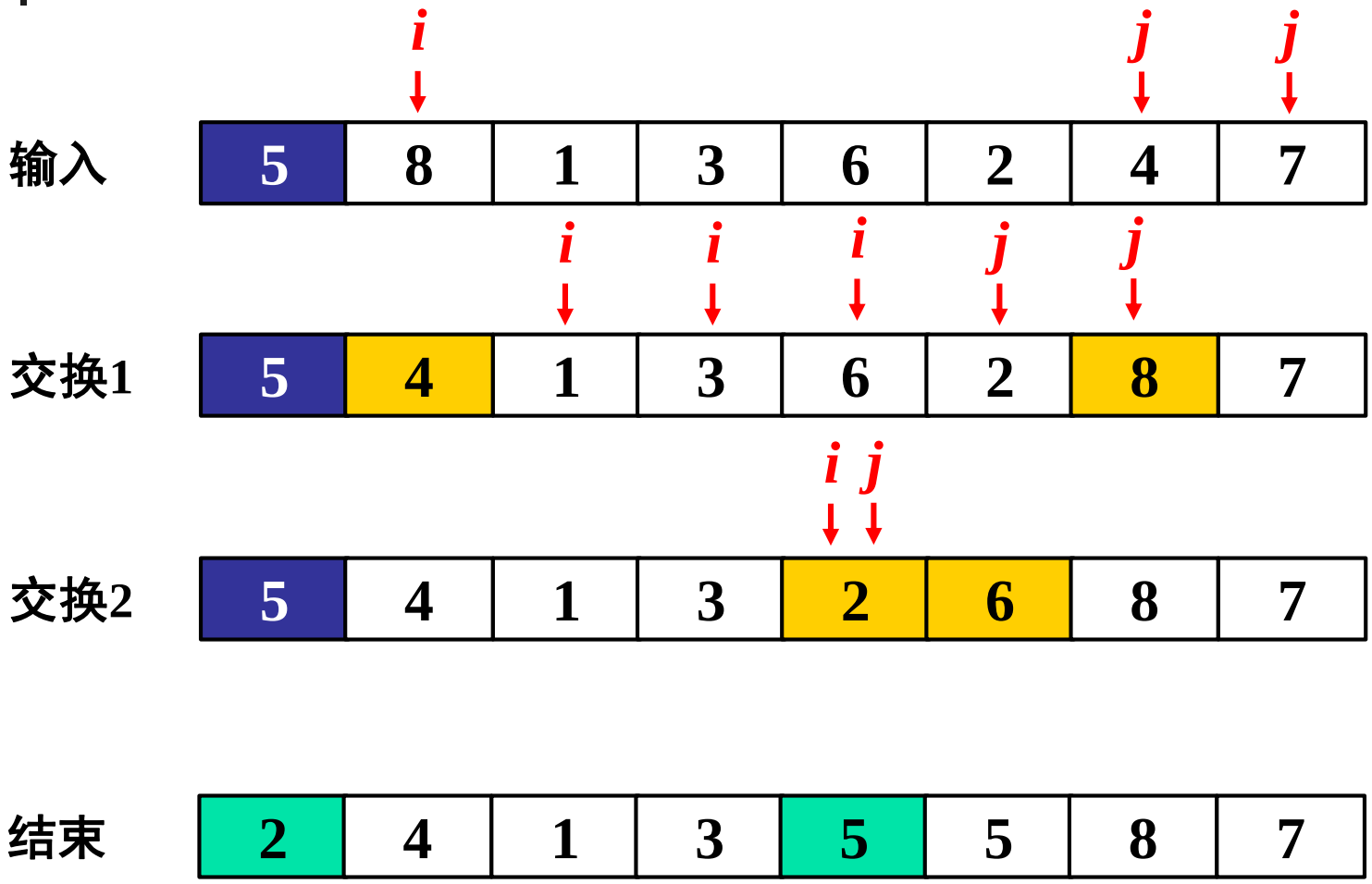
- **划分**：选定一个记录作为轴值，以轴值为基准将整个序列划分为两个子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$，前一个子序列中记录的值均小于或等于轴值，后一个子序列中记录的值均大于或等于轴值；
- **求解子问题**：分别对划分后的每一个子序列递归处理；
- **合并**：由于对子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$的排序是就地进行的，所以合并不需要执行任何操作。

快速排序



- 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。

快速排序一次递归运行


i
j
j
输入
5 8 1 3 6 2 4 7
i
i
i
j
j
交换1
5 4 1 3 6 2 8 7
i j
交换2
5 4 1 3 2 6 8 7
结束
2 4 1 3 5 5 8 7

快速排序

以第一个记录作为轴值，对待排序序列进行划分的过程为：

1. 初始化：取第一个记录作为基准，设置两个参数i，j分别用来指示将要与基准记录进行比较的左侧记录位置和右侧记录位置，也就是本次划分的区间；
2. 右侧扫描过程：将基准记录与j指向的记录进行比较，如果j指向记录的关键码大，则j前移一个记录位置。重复右侧扫描过程，直到右侧的记录小（即反序）
3. 左侧扫描过程：将基准记录与i指向的记录进行比较，如果i指向记录的关键码小，则i后移一个记录位置。重复左侧扫描过程，直到左侧的记录大（即反序）
4. 交换：若i＜j，则将i指向与j指向的记录进行交换；
5. 重复2、3、4步，直到i与j指向同一位置，即基准记录最终的位置，然后将基准记录与i（或j）指向的记录交换。

一次划分算法伪代码

```
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--;
          }
     }
     return i;     // i为轴值记录的最终位置


}
```

快速排序算法伪代码

```
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
    }
}
```

快速排序时间的最好情况

- 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：

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

因此，时间复杂度为$O(n\log n)$。

快速排序时间的最坏情况

- 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）。此时，必须经过n-1次递归调用才能把所有记录定位，而且第i趟划分需要经过n-i次关键码的比较才能找到第i个记录的基准位置，因此，总的比较次数为：

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

因此，时间复杂度为$O(n^2)$。

快速排序时间的平均情况

- 在平均情况下，设基准记录的关键码第k小($1 \leqslant k \leqslant n$)，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$O(n\log n)$。

快速排序复杂度分析小结

- 快速排序算法的性能取决于划分的对称性。
- 快速排序时间与划分是否对称有关，最坏情况一边*n*-1个，一边1个。
  - 如不对称，则：$T(n)=\begin{cases}O(1) & n\le 1\\ T(n-1)+O(n) & n>1\end{cases}$

    解得 $T(n)=O(n^2)$
  - 最好情况，对称，则：$T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$

    解得 $T(n)=O(n\log n)$

结论：
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$O(\log n)$

改进的快速排序

在快速排序算法的每一步中，当数组还没有被划分时，可以在数组*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。

```
int RandomizedPartition (int r[ ], int first, int end){
    int i = Random(first, end);
    Swap(r[i], r[first]);
    return Partition (a, first, end);
}
```

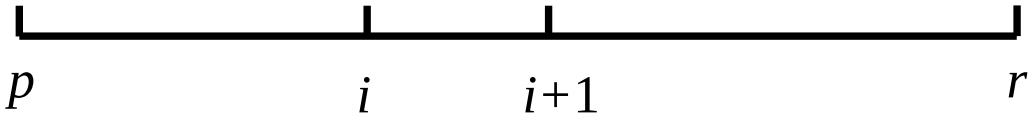
线性时间选择

- **元素选择问题的一般提法：**给定线性序集中n个元素和一个整数k，$1 \leqslant k \leqslant n$，要求找出这$n$个元素中第$k$小的元素。
  - 即如果将这个n个元素依基线排列时，排在第k个位置的元素即为我们所找的元素。
  - 当$k=1$时，为最小元素；
  - 当$k=n$时，为最大元素；
  - 当$k=(n+1)/2$，为中位数。

线性时间选择

```
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);

}
```



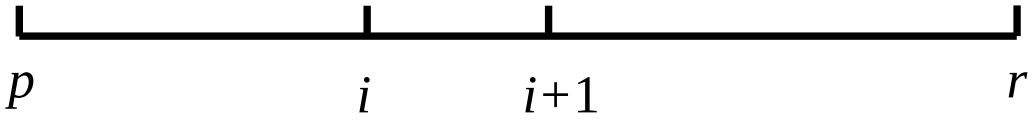
如果(i-p+1 ≥k)，则$a[p:r]$第k小元素落在子数组$a[p:i]$中

$a[p:i]$个数

线性时间选择

```
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);

}
```



如果($k > i-p+1$)，则k落在$a[i+1:r]$是其中第$k-(i-p+1)$小元素

线性时间选择

```
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);

}
```

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。

改进分治算法的途径2：减少子问题数

中国MOOC 3.7 改进分治算法的途径1：减少子问题数

大整数的乘法

- 在复杂性计算时，都将加法和乘法运算当作基本运算处理，即加、乘法时间为常数。但上述假定仅在参加运算整数能在计算机表示范围内直接处理时才合理。

- 那么我们处理很大的整数时，怎么办？

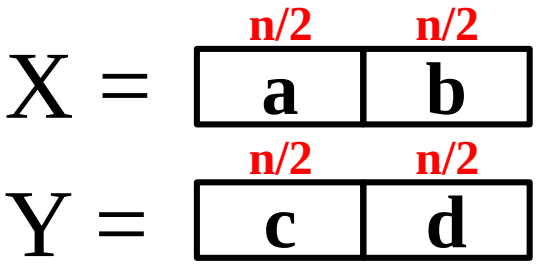
- 要精确地表示大整数，并在计算结果中精确到所有位数，就必须用软件方法实现。

大整数的乘法(cont.)

问题： 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：$O(n^2)$ ✖ 效率太低了！！！

◆分治法：



假设：X和Y都是n位二进制整数

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

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

复杂度分析

则计算X，Y，需4次$n/2$位整数乘法（ac，ad，bc和bd）

需3次小于2n位的整数加

需2次移位(2^n, $2^{n/2}$)

} 共需$O(n)$

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

简化设n为2的幂

$$T(1)=O(1)$$

$$T(2)=4O(1)+O(2) \rightarrow 4O(1)$$

$$T(4)=4(4O(1)+O(2))+O(4) \rightarrow 4^2O(1)$$

$$T(8)=4(4(4O(1)+O(2))+O(4))+O(8) \rightarrow 4^3O(1)$$

复杂度分析

$$T(16)=\cdots \rightarrow 4^4 O(1)$$

$$\cdots$$

$$T(2^x)=\cdots \rightarrow 4^x O(1)$$

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

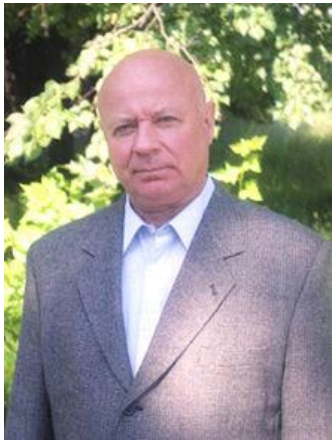
➔ $T(n)=O(n^2)$ ✖ **没有改进！！！**

大整数的乘法(cont.)

Karatsuba trick

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

$$ad + bc = (a-b)(d-c)+ac+bd$$
$$= (a+b)(d+c)-ac-bd$$



Anatoly Karatsuba

大整数的乘法(cont.)

为了降低时间复杂度，必须减少乘法的次数！！！

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

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

对于1式：需3次*n*/2位整数乘法((a-b)(d-c),ac,bd)
需6次加、减
需2次移位

大整数的乘法(cont.)

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+b, c+d可能得到$n/2+1$位的结果，使问题的规模变大，故不选择第2种方案。

大整数的乘法(cont.)

- 还有没有更快的方法？ $T(n)=O(n^{\log 3}) = O(n^{1.59})$

- 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。

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$

是否能找到线性时间算法？目前为止还没有结果。

Strassen矩阵乘法

A和B的乘积矩阵C中的元素C[i,j]定义为:

$$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$$

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

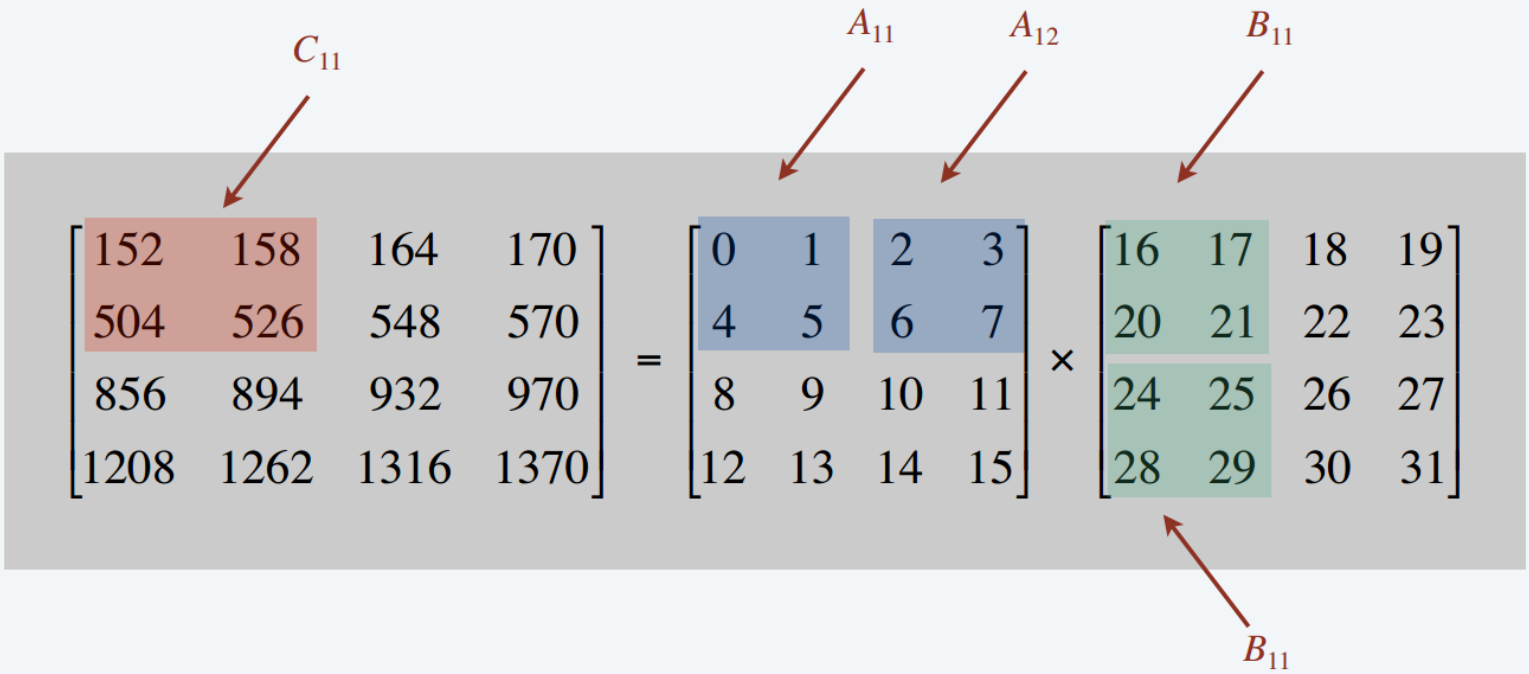
Strassen矩阵乘法

A和B的乘积矩阵C中的元素C[i,j]定义为:

$$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$$

分析：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n^2个元素所需的计算时间为$O(n^3)$

Strassen矩阵乘法



$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Strassen矩阵乘法

分治法：

使用与上例类似的技术，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：

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

由此可得：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

复杂度分析

1）n=2，子矩阵阶为1，8次乘和4次加，直接求出；
2）子矩阵阶大于2，为求子矩阵积可继续分块，直到子矩阵阶降为2。

此想法就产生了一个分治降阶递归算法。

两个n阶方阵的积➔8个n/2阶方阵积和4个n/2阶方阵加。

可在$O(n^2)$时间内完成

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

所以 $T(n)=O(n^3)$**，与原始定义计算相比并不有效。**

复杂度分析(cont.)

原因是此法没有减少矩阵的乘法次数！！！

下面从计算2个2阶方阵乘开始，研究减少乘法次数(小于8次)

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

7次乘

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

复杂度分析(cont.)

所以 需要7次乘法
18次加减法 → $T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$

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

Strassen矩阵乘法

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?

A. Yes! [Strassen 1969] $\Theta(n^{\log_2 7}) = O(n^{2.807})$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Multiply two 3-by-3 matrices with 21 scalar multiplications?

A. Unknown. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]

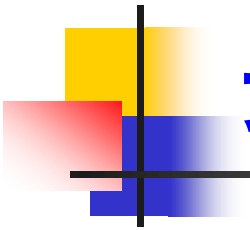
- Two 20-by-20 matrices with 4,460 scalar multiplications. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar multiplications. $O(n^{2.7801})$
- A year later. $O(n^{2.7799})$
- December 1979. $O(n^{2.521813})$
- January 1980. $O(n^{2.521801})$

Strassen矩阵乘法

year	algorithm	order of growth
?	brute force	$O(n^3)$
1969	Strassen	$O(n^{2.808})$
1978	Pan	$O(n^{2.796})$
1979	Bini	$O(n^{2.780})$
1981	Schönhage	$O(n^{2.522})$
1982	Romani	$O(n^{2.517})$
1982	Coppersmith-Winograd	$O(n^{2.496})$
1986	Strassen	$O(n^{2.479})$
1989	Coppersmith-Winograd	$O(n^{2.376})$
2010	Strother	$O(n^{2.3737})$
2011	Williams	$O(n^{2.3727})$
?	?	$O(n^{2+\varepsilon})$

改进分治算法的途径3：增加预处理

中国MOOC 3.8 改进分治算法的途径2：增加预处理



最接近点对问题

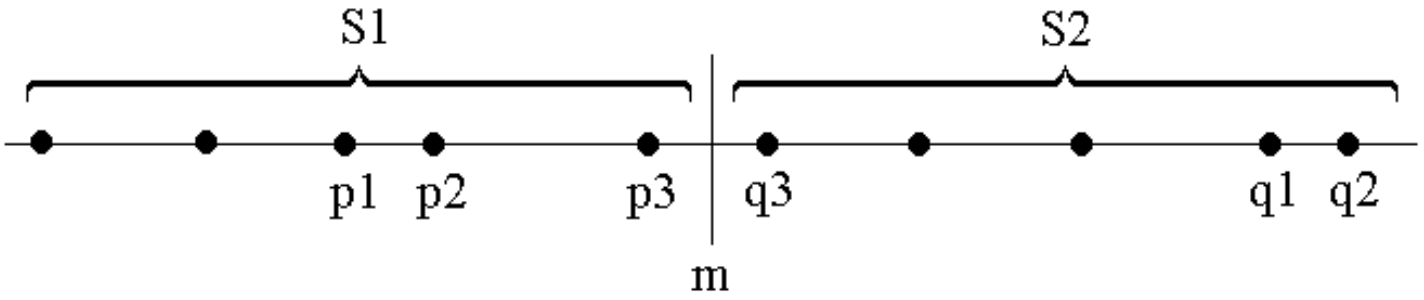
问题： 给定平面上n个点的集合S，找其中的一对点，使得在n个点组成的所有点对中，该点对间的距离最小。

最接近点对问题

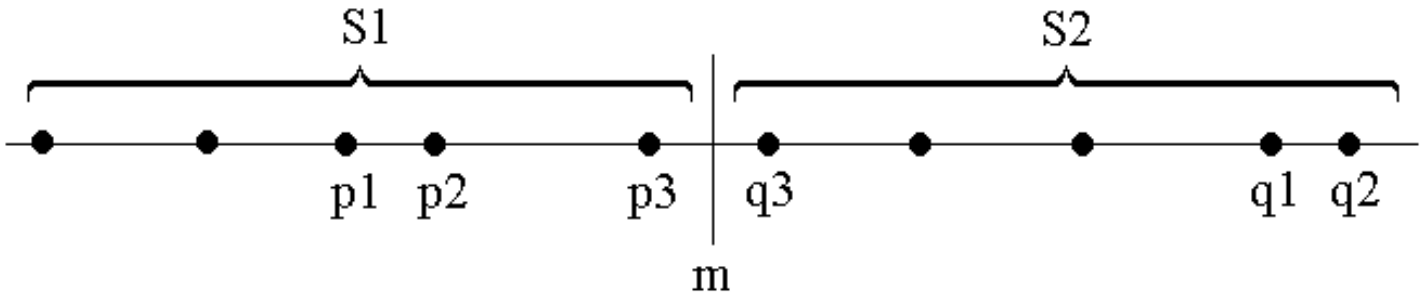
问题： 给定平面上n个点的集合S，找其中的一对点，使得在n个点组成的所有点对中，该点对间的距离最小。

最接近点对问题

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 $x_1, x_2, \cdots x_n$。最接近点对即为这n个实数中相差最小的2个实数。



最接近点对问题



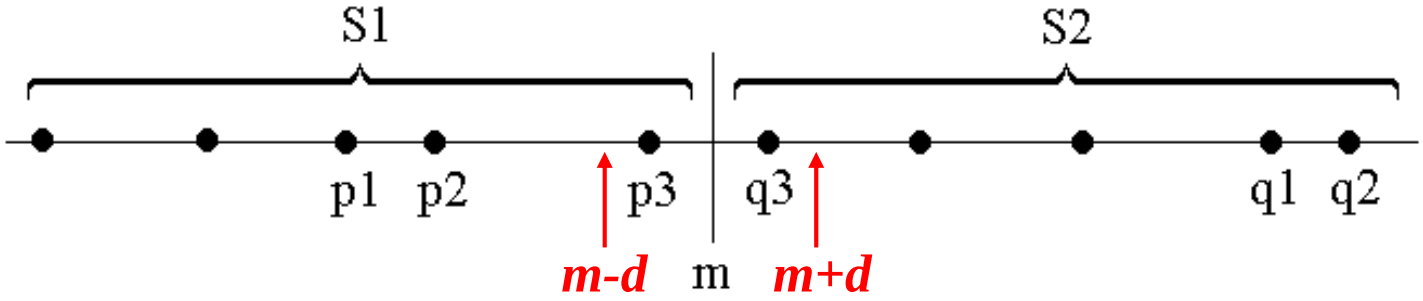
假设： 用x轴上某个点m将S划分为2个子集S1和S2 ，基于**平衡子问题**的思想，用S中各点坐标的中位数来作分割点。

递归地，在S1和S2上找出其最接近点对{p1,p2}和{q1,q2}，并设**d=min{|p1-p2|,|q1-q2|}**，S中的最接近点对或者是{p1,p2}，或者是{q1,q2}，或者是某个{p3,q3}，其中p3∈S1且q3∈S2。

最接近点对问题

能否在线性时间内找到p3, q3？

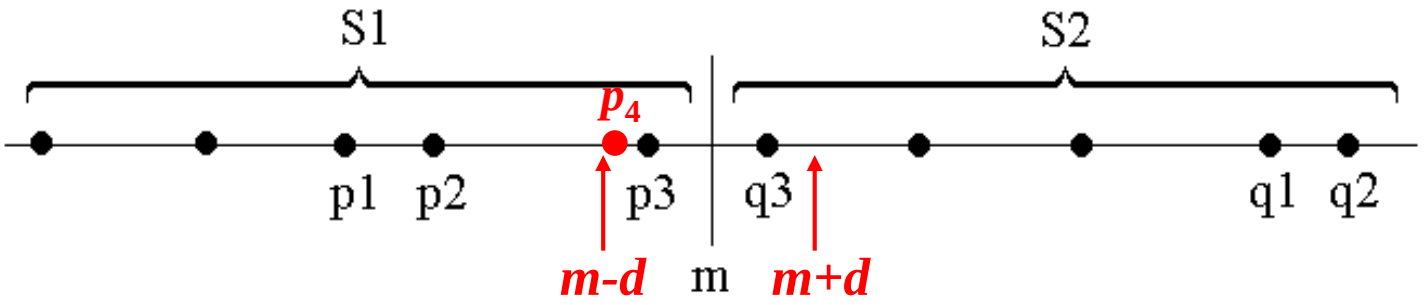
- 如果S的最接近点对是{p3,q3}，即|p3-q3|<d，则p3和q3两者与m的距离不超过d，即**p3∈(m-d,m]，q3∈(m,m+d]**。



最接近点对问题

能否在线性时间内找到p3, q3？

- 由于在S1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S1和S2的分割点，因此(m-d,m]中至多包含S中的一个点。由图可以看出，**如果(m-d,m]中有S中的点，则此点就是S1中最大点。**



最接近点对问题

能否在线性时间内找到p3, q3？

- 因此，用线性时间就能找到区间(m-d,m]和(m,m+d]中所有点，即p3和q3。**从而用线性时间就可以将S1的解和S2的解合并成为S的解。**

$$d = \min(d_{S1}, d_{S2}, q_3 - p_3)$$

最接近点对问题

复杂度分析：

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

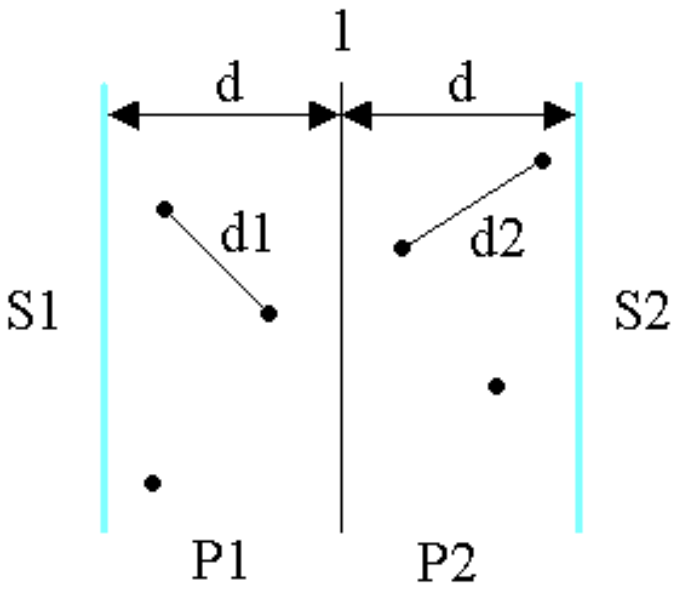
➔ ***T(n)=O(nlogn)***

看上去比用排序加扫描的算法复杂，然而它可以推广到二维的情形。

最接近点对问题

下面来考虑二维的情形

- 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S1和S2。
- 递归地在S1和S2上找出其最小距离d1和d2，并设d=min{d1,d2}，S中的最接近点对或者是d，或者是某个{p,q}，其中p∈P1且q∈P2。
- **能否在线性时间内找到p, q?**

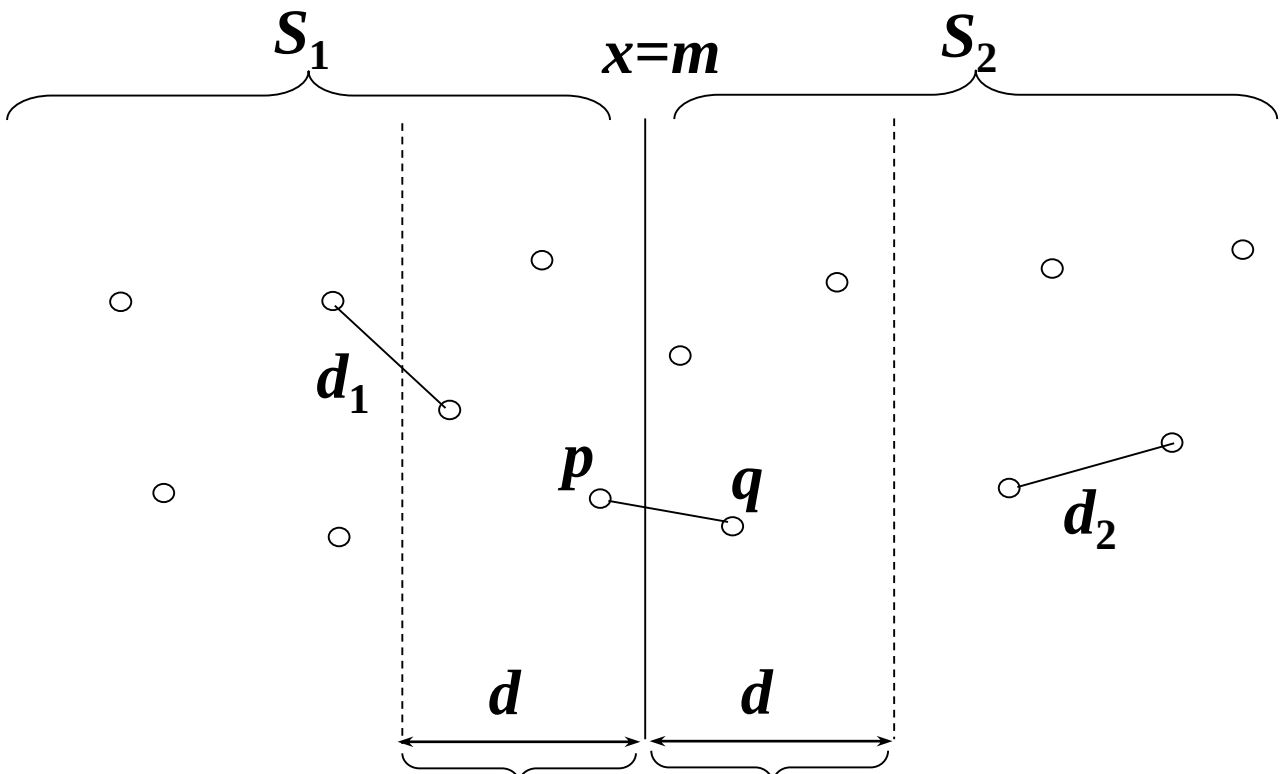


最接近点对问题

能否在线性时间内找到p3, q3？

- 考虑P1中任意一点p，它若与P2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P2中的点一定落在一个d×2d的矩形R中
- 由d的意义可知，P2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

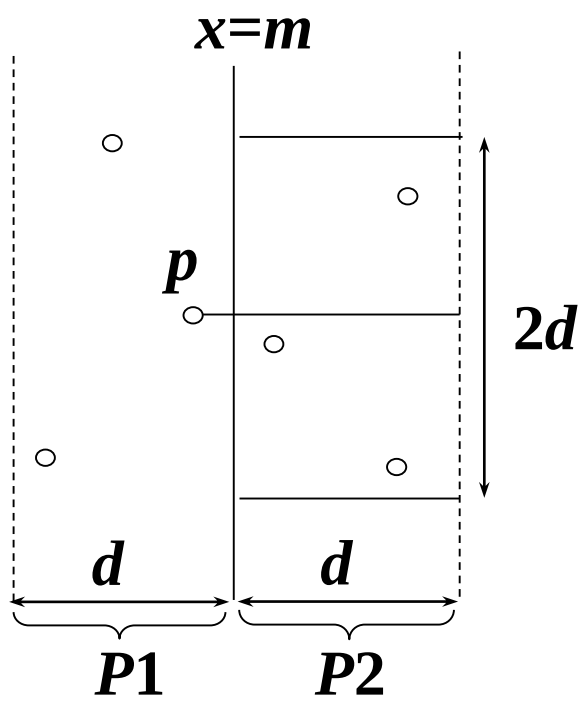
最接近点对问题



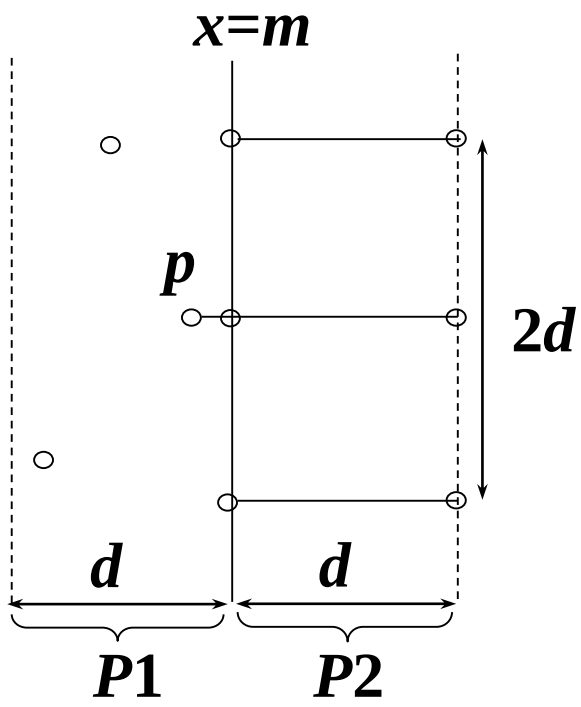
$S_1=\{p \in S \mid x_p \leq m\}$

$S_2=\{q \in S \mid x_q > m\}$

对于点$p \in P1$，需要考察$P2$中的各个点和点p之间的距离是否小于d，显然，$P2$中这样点的y轴坐标一定位于区间$[y-d, y+d]$之间，而且，这样的点不会超过6个。



(a) 包含点q的$d \times 2d$的矩形区域

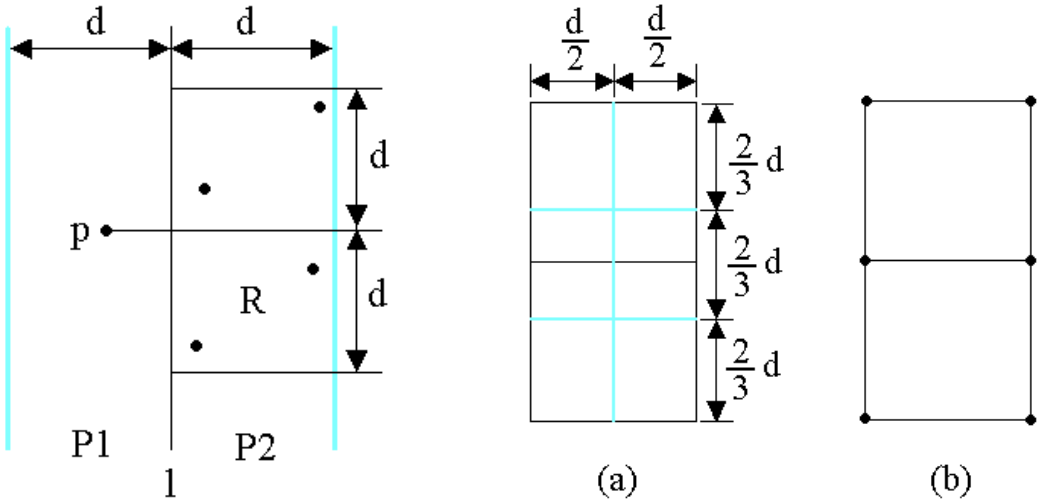


(b) 最坏情况下需要检查的6个点

最接近点对问题

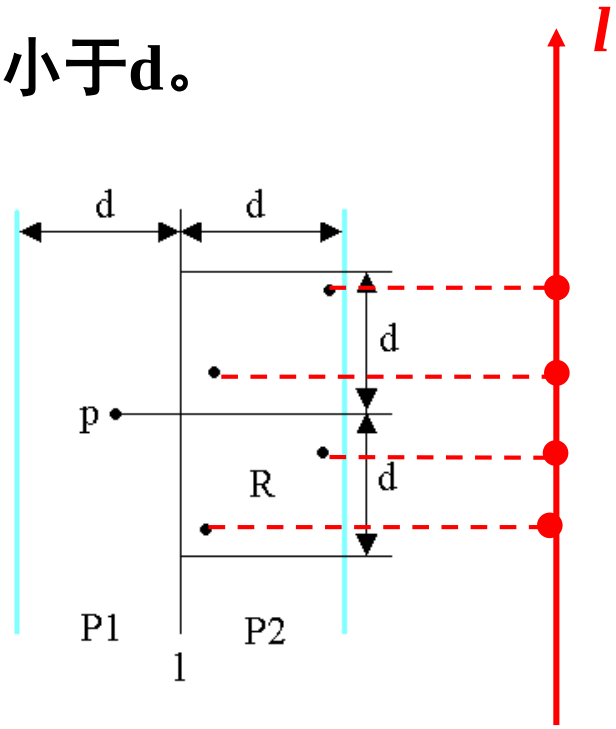
证明： 将矩形R的长为2d的边3等分，将它的长为d的边2等分，由此导出6个(d/2)×(2d/3)的矩形。若矩形R中有多于6个S中的点，则由鸽舍原理易知至少有一个(d/2)×(2d/3)的小矩形中有2个以上S中的点。设u，v是位于同一小矩形中的2个点，则 $(x(u)-x(v))^2+(y(u)-y(v))^2 \le (d/2)^2+(2d/3)^2=\frac{25}{36}d^2$

distance(u,v)<d。这与d的意义相矛盾。



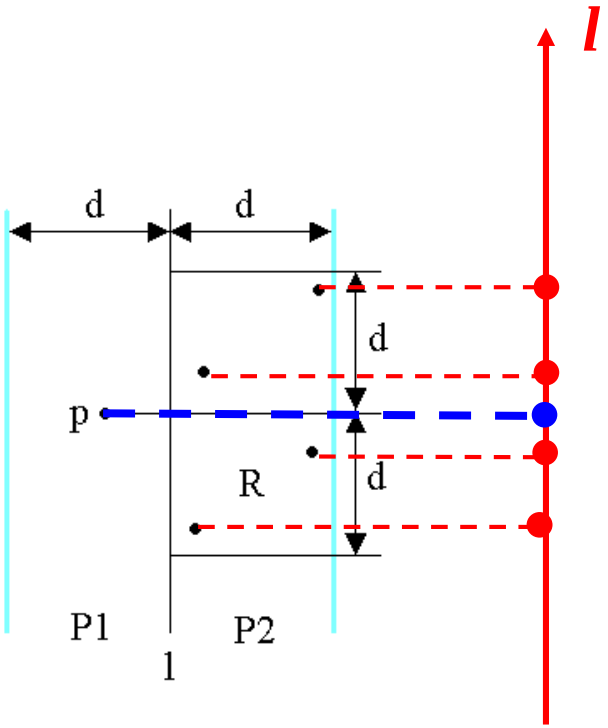
最接近点对问题

- 要检查哪6个点?
  - 将p和P2中所有S2的点投影到垂直线l上。
  - 投影点距p在l上投影点的距离小于d。
  - 这种投影点最多只有6个。



最接近点对问题

- 快速检查6个点的方法
  - 将P1和P2中所有S中点按其y坐标排好序（预处理）
  - 对P1中所有点，对排好序的点列作一次扫描，
  - 对P1中每一点最多只要检查P2中排好序的相继6个点。



最接近点对问题

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*)**