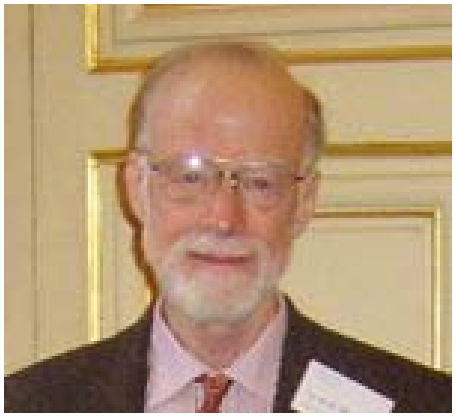
3 分治策略

Divide and Conquer

引例：快速排序



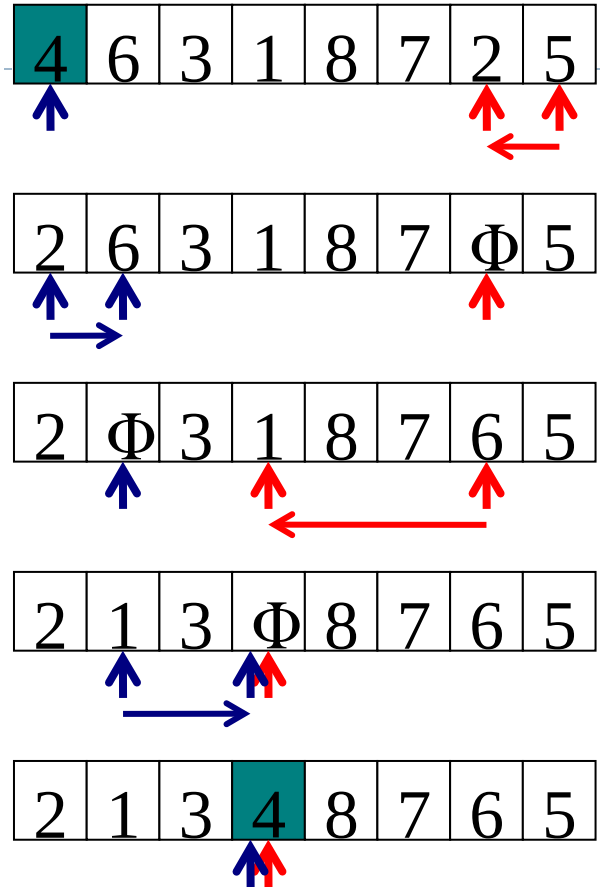
快速排序是一个非常流行而且高效的算法，其平均时间复杂度为Θ(nlogn). 其优于合并排序之处在于它在原位上排序，不需要额外的辅助存贮空间(合并排序需Θ(n)的辅助空间)。Charles A. R. Hoare 1960 年发布了使他闻名于世的快速排序算法(Quicksort)，这个算法也是当前世界上使用最广泛的算法之一，当时他供职于伦敦一家不大的计算机生产厂家。1980 年，Hoare 被授予 Turing 奖，以表彰其在程序语言定义与设计领域的根本性的贡献。在 2000 年，Hoare 因其在计算机科学和教育方面的杰出贡献被英国皇家封为爵士。

将 A[1...8]={4, 6, 3, 1, 8, 7, 2, 5} 按照非降序方式进行排序

- SPLIT:以待排序数组的首元素作为基准元素，将待排序数组分成左右两个子数组。使得左边子数组中的元素都小于等于基准元素；右边子数组中的元素都大于等于基准元素。
- 对左、右子数组进行相同的操作。

Algorithm: SPLIT(A[low,...high])
输入：数组A[low,...high]
输出：用A[low]作基准元素划分后的数组A
及基准元素新的位置w

1. x←A[low]
2. while (low<high)
3. while (low<high && A[high]>x) --high;
4. A[low] ←A[high]
5. while (low<high && A[low]≤x) ++low;
6. A[high] ←A[low]
7. end while
8. A[low] ← x
9. w←low
10. return A and w //新数组A与x的新位置w



```
Algorithm: QUICKSORT(A[low…high])
输入: n个元素的数组A[low…high]
输出：按非降序排列的数组A[low…high]
1.  if low<high then
2.     w ← SPLIT(A[low…high])  {w为基准元素A[low]的新位置}
3.     quicksort(A, low, w-1)
4.     quicksort(A, w+1, high)
5.  end if
```

时间复杂度分析

理想情形：每次SPLIT后得到的左右子数组规模相当，因此有：

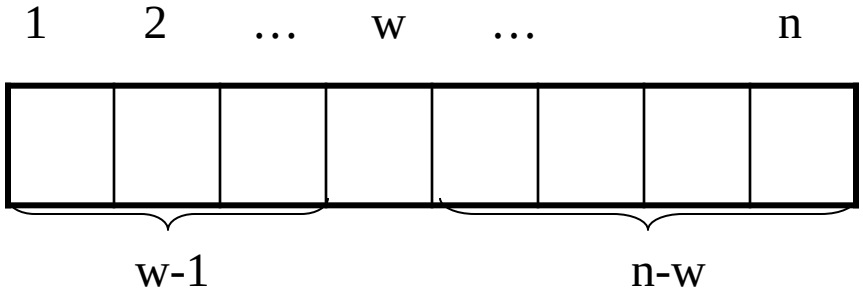
$$T(n)=\begin{cases}O(1) & if\ n=1\\ 2T(n/2)+\Theta(n) & if\ n>1\end{cases}\longrightarrow T(n)=\Theta(n\log n)$$

最差情形(已经排好序或是逆序的数组)：每次SPLIT后，只得到左或是右子数组，因此有：

$$T(n)=\begin{cases}O(1) & if\ n=1\\ T(n-1)+\Theta(n) & if\ n>1\end{cases}\longrightarrow T(n)=\Theta(n^2)$$

平均情形：

我们用$C(n)$ 表示对一个n个元素的数组进行快速排序所需要的总的比较次数。



因此，我们有：

$$C(n)=(n-1)+\frac{1}{n}\sum_{w=1}^{n}\left(C(w-1)+C(n-w)\right)$$

$$\because \sum_{w=1}^{n}C(n-w)=C(n-1)+C(n-2)+\cdots+C(0)=\sum_{w=1}^{n}C(w-1)$$

$$\therefore C(n)=(n-1)+\frac{2}{n}\sum_{w=1}^{n}C(w-1)$$

$$n \cdot C(n) = n(n-1) + 2\sum_{w=1}^{n} C(w-1) \ldots\ldots (a)$$

↓ n-1替换n

$$(n-1)C(n-1) = (n-1)(n-2) + 2\sum_{w=1}^{n-1} C(w-1) \ldots\ldots (b)$$

(a)-(b),并适当变换

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

令 $D(n) = \frac{C(n)}{n+1}$ ↓

$$D(n) = D(n-1) + \frac{2(n-1)}{n(n+1)}, D(1) = 0$$

↓

$$D(n) = 2\sum_{j=1}^{n} \frac{j-1}{j(j+1)} = 2\sum_{j=1}^{n} \frac{2}{(j+1)} - 2\sum_{j=1}^{n} \frac{1}{j}$$

$$= 4\sum_{j=2}^{n+1} \frac{1}{j} - 2\sum_{j=1}^{n} \frac{1}{j} = 2\sum_{j=1}^{n} \frac{1}{j} - \frac{4n}{n+1} = \Theta(\log n)$$

$$\therefore C(n) = (n+1)D(n) = \Theta(n \log n)$$

分治策略的思想

- 把规模较大的问题分解为若干个规模较小的子问题，这些子问题相互独立且与原问题同类；（该子问题的规模减小到一定的程度就可以容易地解决）
- 依次求出这些子问题的解，然后把这些子问题的解组合起来得到原问题的解。
- 由于子问题与原问题是同类的,故分治法可以很自然地应用递归。

使用分治策略的算法设计模式

```
divide_and_conquer(P)
{
    if(|P|<=n0)
        direct_process(P);
    else
    {
        divide P into smaller subinstances P1,P2,…,Pa
        for(int i=1;i<=a;i++)
            yi=divide_and_conquer(Pi);
        merge(y1,y2,…,ya);
    }
}
```

使用分治策略的算法的时间复杂度分析

- 从分治法的一般设计模式可以看出，用它设计出的算法通常可以是递归算法。因而，算法的时间复杂度通常可以用递归方程来分析。
- 假设算法将规模为n的问题分解为a(a>=1)个规模为n/b(b>1)的子问题解决。分解子问题以及合并子问题的解耗费的时间为s(n)，则算法的时间复杂度可以递归表示为：

$$T(n)=\begin{cases} c & ,\ n <= n_0 \\ aT(n/b)+s(n) & ,\ n > n_0 \end{cases}$$

- 回顾Master Theorem

合并排序Merge Sort

例：给定数组A [1…8]=

8	4	3	1	6	2	9	7

1. 将其分分成左右两个子数组:

8	4	3	1

6	2	9	7

2. 对子数组进行排序。

3. 对排序后的子数组进行合并 :两个已排序的子数组用A[p…q]和A[q+1…r]表示. 设两个指针s和t，初始时各自指向A[p]和A[q+1],再设一空数组B[p…q, q+1…r]做暂存器，比较元素A[s]和A[t], 将较小者添加到B , 然后移动指针,

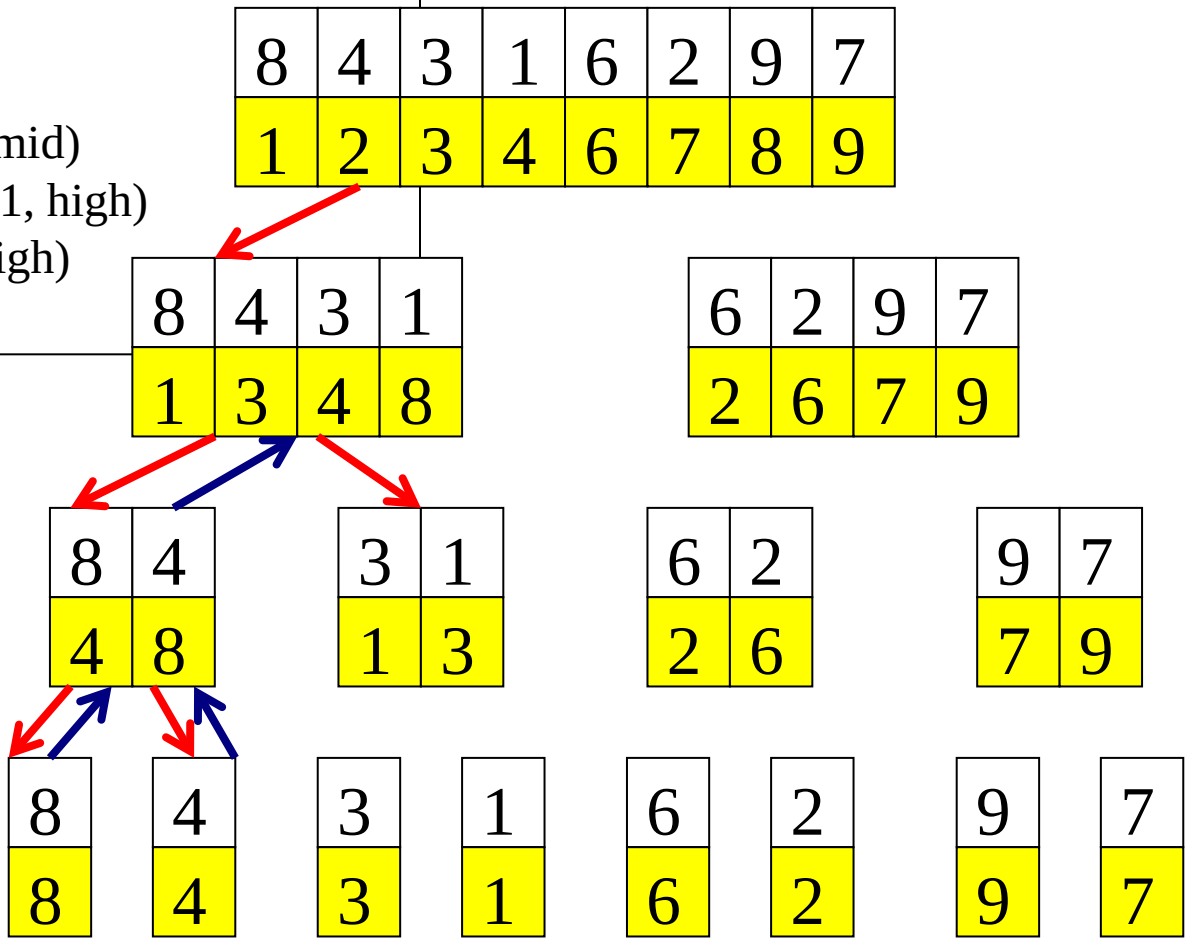
若A[s]较小，则s+1，否则t+1,

直到s=q+1 或 t=r+1 为止

将剩余元素A[t…r] 或 A[s…q] 拷贝到数组B, 然后令A←B.

Algorithm: MERGE(A, p, q, r)
输入：数组A[p...q]和A[q+1...r], 各自按升序排列
输出：将A[p...q]和A[q+1...r]合并成一个升序排序的新数组
1. s←p; t←q+1; k←p; {s, t, p 分别指向A[p...q],A[q+1...r]和B}
2. while s≤q and t≤r
3. if A[s] ≤A[t] then
4. B[k]←A[s]
5. s ←s+1
6. else
7. B[k]←A[t]
8. t←t+1
9. end if
10. k←k+1
11.end while
12. if s=q+1 then B[k...r] ←A[t...r]
13. else B[k...r] ←A[s...q]
14. end if
15. A[p...r] ←B[p...r]

Algorithm: MERGESORT(A[low…high])
输入：待排序数组A[low,...high]
输出：A[low…high]按非降序排列
1. if low<high then
2. mid←⌊(low+high)/2⌋
3. MERGESORT(A, low, mid)
4. MERGESORT(A, mid+1, high)
5. MERGE(A, low, mid, high)
6. end if



时间复杂度分析

最小比较次数

$$C(n)=\begin{cases}0 & if\ n=1\\ 2C(n/2)+n/2 & if\ n\geq 2\end{cases}\quad\Rightarrow\quad C(n)=\frac{n\log n}{2}$$

最大比较次数

$$C(n)=\begin{cases}0 & if\ n=1\\ 2C(n/2)+n-1 & if\ n\geq 2\end{cases}\quad\Rightarrow\quad C(n)=n\log n-n+1$$

矩阵乘法

- 设A,B是两个n×n的矩阵，求C=AB.
- 方法1: 直接相乘法
- 方法2: 分块矩阵法(直接应用分治策略)
- 方法3: Strassen算法(改进的分治策略)

方法1：直接相乘

$$C=[c_{ij}]_{i=1,2,\cdots,n;\ j=1,2,\cdots,n} \qquad c_{ij}=\sum_{k=1}^{n} a_{ik}b_{kj}$$

时间复杂度分析：

假设每做一次标量乘法耗费时间为m,每做一次标量加法耗费时间为a,那么直接相乘算法的时间复杂度为：

$$T(n)=n^3m+n^2(n-1)a=\Theta(n^3)$$

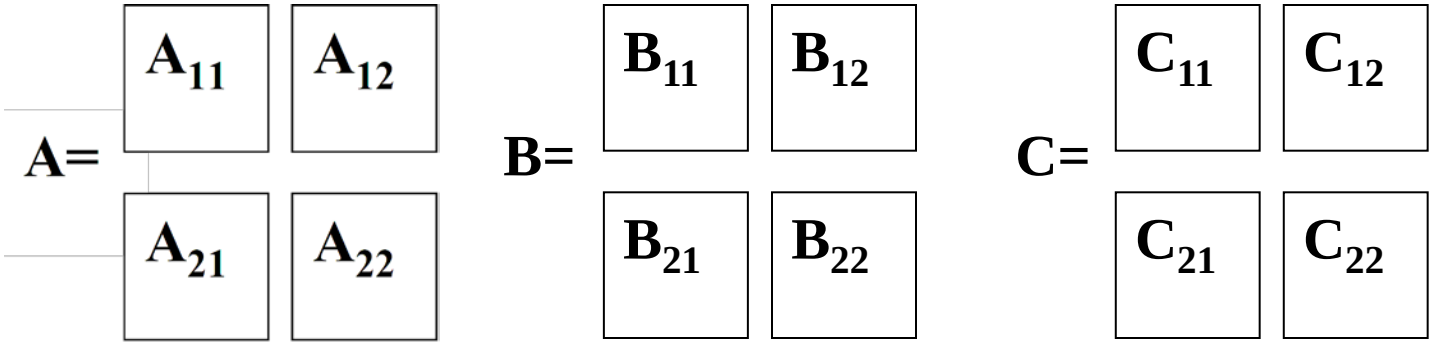
矩阵分块

$$A=\begin{pmatrix}1&1&2&2\\1&2&3&3\\3&3&1&1\\2&1&2&2\end{pmatrix}\qquad B=\begin{pmatrix}2&1&1&2\\3&2&2&3\\1&3&1&1\\2&1&2&4\end{pmatrix}$$

$$A\square B=\begin{pmatrix}1&1&2&2\\1&2&3&3\\3&3&1&1\\2&1&2&2\end{pmatrix}\square\begin{pmatrix}2&1&1&2\\3&2&2&3\\1&3&1&1\\2&1&2&4\end{pmatrix}$$

$$A\square B=\left(\begin{array}{cc:cc}1&1&2&2\\1&2&3&3\\\hdashline3&3&1&1\\2&1&2&2\end{array}\right)\square\left(\begin{array}{cc:cc}2&1&1&2\\3&2&2&3\\\hdashline1&3&1&1\\2&1&2&4\end{array}\right)$$

方法2:分块矩阵法(直接应用分治策略)



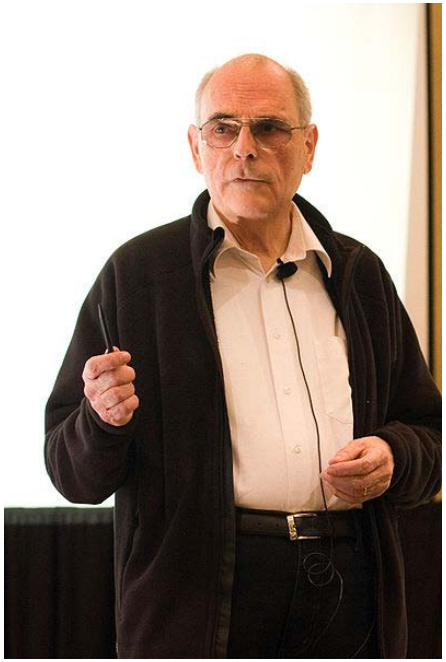
$$C_{11}=A_{11}B_{11}+A_{12}B_{21}$$
$$C_{12}=A_{11}B_{12}+A_{12}B_{22}$$
$$C_{21}=A_{21}B_{11}+A_{22}B_{21}$$
$$C_{22}=A_{21}B_{12}+A_{22}B_{22}$$

$$T(n)=\begin{cases} m & if \quad n=1 \\ 8T(n/2)+4(n/2)^2 a & if \quad n\geq 2 \end{cases}$$

4次加法

$$T(n)=\Theta(n^3)$$

Strassen算法



Volker Strassen giving the Knuth Prize lecture at SODA 2009.

Strassen was born on April 29, 1936, in Germany. In 1969, Strassen shifted his research efforts towards the analysis of algorithms with a paper on Gaussian elimination, introducing Strassen's algorithm, the first algorithm for performing matrix multiplication faster than the $O(n^3)$ time bound that would result from a naive algorithm. In the same paper he also presented an asymptotically-fast algorithm to perform matrix inversion, based on the fast matrix multiplication algorithm. This result was an important theoretical breakthrough, leading to much additional research on fast matrix multiplication, and despite later theoretical improvements it remains a practical method for multiplication of dense matrices of moderate to large sizes.

—From Wikipedia, the free encyclopedia

引入下列 $M_i(i=1,2,\cdots,7)$：

$M_1=(A_{12}-A_{22})(B_{21}+B_{22})$

$M_2=(A_{11}+A_{22})(B_{11}+B_{12})$

$M_3=(A_{11}-A_{21})(B_{11}+B_{12})$

$M_4=(A_{11}+A_{12})B_{22}$

$M_5=A_{11}(B_{12}-B_{22})$

$M_6=A_{22}(B_{21}-B_{11})$

$M_7=(A_{21}+A_{22})B_{11}$

则有：$C_{11}=M_1+M_2-M_4+M_6$，$C_{12}=M_4+M_5$，

$C_{21}=M_6+M_7$，$C_{22}=M_2-M_3+M_5-M_7$

$$T(n)=\begin{cases} m & if \quad n=1 \\ 7T(n/2)+18(n/2)^2 a & if \quad n\geq 2 \end{cases}$$

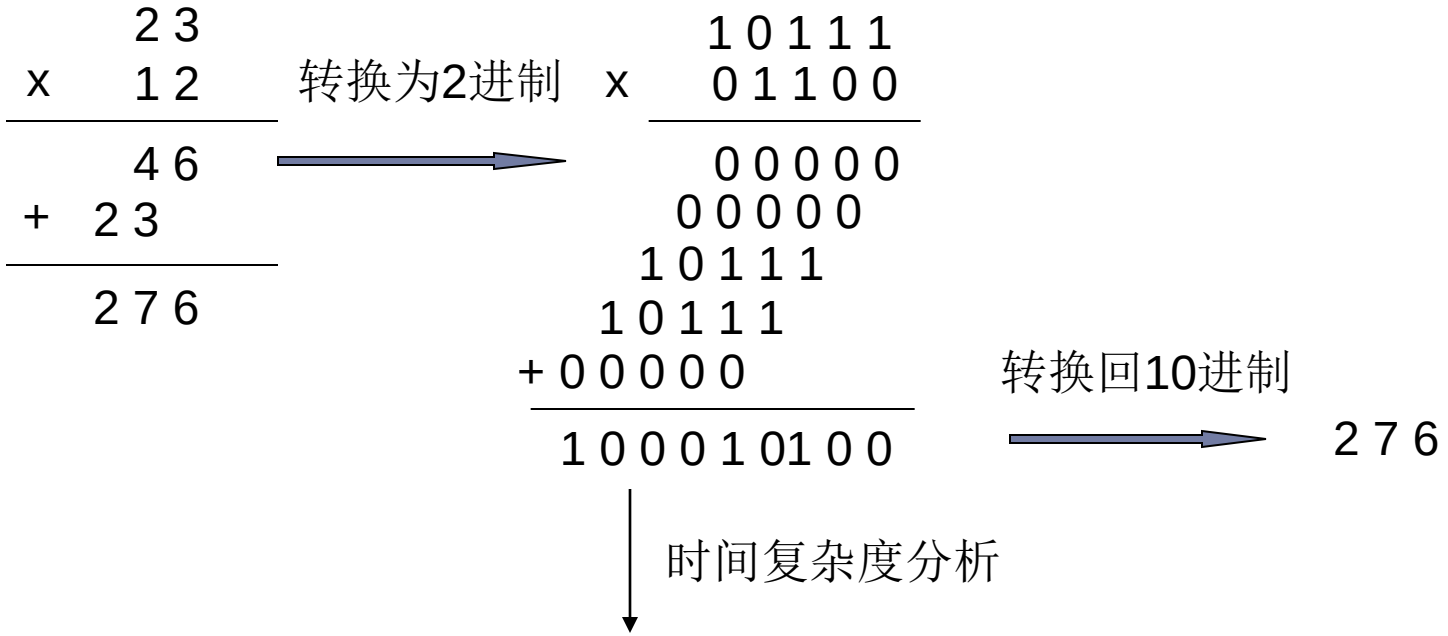
18次加法

$$T(n)=\Theta(n^{\log_b^a})=\Theta(n^{\log_2^7})=\Theta(n^{2.81})$$

大整数相乘

- 通常，在分析算法的计算复杂度时，都将加法和乘法运算当作基本运算来处理，即将执行一次加法或乘法运算所需要的计算时间当作一个常数，该常数仅仅取决于计算机硬件处理速度。
- 然而，这个假定仅仅在参加运算的整数处于一定范围内时才是合理的。这个整数范围取决于计算机硬件对整数的表示。
- 在某些情况下，要处理很大的整数，它无法在计算机硬件能直接表示的整数范围内进行处理。这时候，就必须使用软件的方法来实现大整数的算术运算。

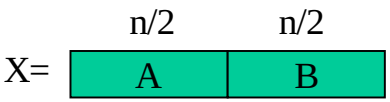
- 问题：设有两个n bit位的二进制整数X, Y，要计算XY.

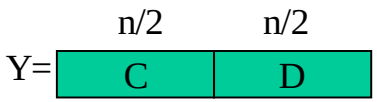


将位(bit)的乘法或加法当作基本运算，则逐位相乘算法的时间复杂度为：

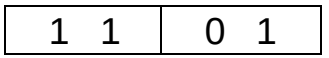
$$T(n) = \Theta(n^2)$$

使用分治策略来求解之





例

$X =$ 

$A = (11)_2 = 3$

$B = (01)_2 = 1$

我们有：

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

例

$$X = 3 \cdot 2^{4/2} + 1 = 13$$

下式成立：

$$XY = (A \cdot 2^{n/2} + B)(C \cdot 2^{n/2} + D)$$

$$= A \cdot C2^{n} + (A \cdot D + C \cdot B)2^{n/2} + B \cdot D$$

时间复杂性分析

$$XY = (A \cdot 2^{n/2} + B)(C \cdot 2^{n/2} + D)$$

$$= A \cdot C \cdot 2^{n} + (A \cdot D + C \cdot B) \cdot 2^{n/2} + B \cdot D$$

(1)4次$n/2$ bit位数的乘法($A \cdot C, A \cdot D, C \cdot B, B \cdot D$).//$4T(n/2)$

(2)$A \cdot C$左移n位($A \cdot C \cdot 2^{n}$). //$\Theta(n)$

(3)求$A \cdot D + C \cdot B$所作的n位加法. //$\Theta(n)$

(4)$A \cdot D + C \cdot B$左移$n/2$位($(A \cdot D + C \cdot B)2^{n/2}$). //$\Theta(n)$

(5)两个加法($A \cdot C \cdot 2^{n} + (A \cdot D + C \cdot B) \cdot 2^{n/2} + B \cdot D$). //$\Theta(n)$

$$T(n)=\begin{cases} a & if\ n=1 \\ 4T(n/2)+\Theta(n) & if\ n>1 \end{cases} \longrightarrow T(n)=\Theta\left(n^2\right)$$

如何降低时间复杂性？

观察：$A\cdot D+B\cdot C=(A+B)(C+D)-AC-BD$

所以：

$$\begin{aligned} XY &= (A\cdot 2^{n/2}+B)(C\cdot 2^{n/2}+D) \\ &= A\cdot C\cdot 2^n+(A\cdot D+C\cdot B)\cdot 2^{n/2}+B\cdot D \\ &= A\cdot C\cdot 2^n+((A+B)(C+D)-AC-BD)\cdot 2^{n/2}+B\cdot D \end{aligned}$$

$$T(n)=\begin{cases} a & ,if\ \ n=1 \\ 3T(n/2)+\Theta(n) & ,if\ \ n>1 \end{cases} \longrightarrow T(n)=\Theta(n^{\log 3})=\Theta(n^{1.59})$$

排列问题

已知集合R={$r_1, r_2,..., r_n$}，请设计一个算法生成集合R 中n 个元素的全排列，并给出算法时间复杂性分析。

解：令R_j=R-{r_j}，我们记集合X 中元素的全排列记为perm(X)。
那么(r_j)perm(X)表示在全排列perm(X)的每一种排列前加上前缀r_j所得到的排列。

所以，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。

算法的一个Java实现如下:

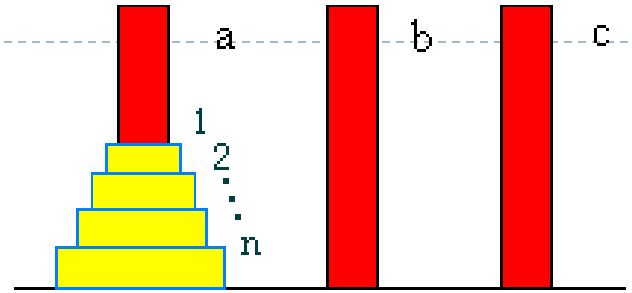
```
public class Perm {
   public static void perm(Object[] a,
                           int k, int m)
{
      if (k == m)
      {
         for (int i = 0; i < a.length; i++) {
            System.out.print(a[i]);
         }
         System.out.println();
      } else
      {
         for (int i = k; i <= m; i++) {
            swap(a, k, i);
            perm(a, k + 1, m);
            swap(a, k, i);
         }
      }
   }
```

```
public static void main(String[] args) {
      Integer[] a = new Integer[6];
      for (int i = 0; i < a.length; i++) {
         a[i] = new Integer(i);
      }
      perm(a, 0, 5);
   }
 public static void swap(Object[] a, int i, int j) {
      Object temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

$$T(n) = n(T(n-1)+b) = \Theta(n \cdot n!)$$

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。




```
void hanoi(int n, int a, int b, int c) {
      if (n > 0){
         hanoi(n-1, a, c, b);  //a→c, b作辅助
         move(a, b);  // a→b
         hanoi(n-1, c, b, a);  //c→b, a作辅助
      }
  }
```

寻找数组的中项和第k小元素

- 给定已排好序(非降序)的数组A[1…n],中项是指其“中间”元素。若n为奇数，则中项为数组中的第(n+1)/2个元素；若n为偶数，则存在两个中间元素，分别为第n/2和第n/2+1个元素，在这种情形下，我们取第n/2个元素作为中项；综上，中项为第$\lceil n/2 \rceil$个最小元素。
- 寻找中项的一个直接的方法：先排序，后取中项。显然，该方法的时间复杂度至少为Ω(nlogn)。能否找到更为高效的方法？
- 寻找中项是寻找第k小元素的一个特列。如果能解决寻找第k小元素的问题，那么当k=$\lceil n/2 \rceil$时，解决的就是寻找中项问题。
- 回顾二分搜索：以中间元素为基准抛弃部分元素，不断减小问题规模。

思路

- 如果数组A中元素的个数(问题的规模)小于一个阈值，那么采用直接的方法(先排序，后查找)寻找第k小元素。
- 否则，将n个元素划分成$\lfloor n/5 \rfloor$组，每组5个元素，如果n不是5的倍数，则排除剩余的元素。(注意：这并不代表第k小元素不可能位于这些排除的元素中，详见后续分析)。
- 对每组元素排序，并取出它们的中项(即第3个元素)。$\lfloor n/5 \rfloor$个中项的中项，我们记为mm。
- 依据mm将数组A划分为三个子数组：$A_1 = \{a|a<mm\}$、$A_2 = \{a|a=mm\}$、$A_3=\{a|a>mm\}$。
- 判断第k小元素可能在哪一个子数组中出现：如果在A_2中出现，则已经找到；否在，在A_1或A_3上进行递归。

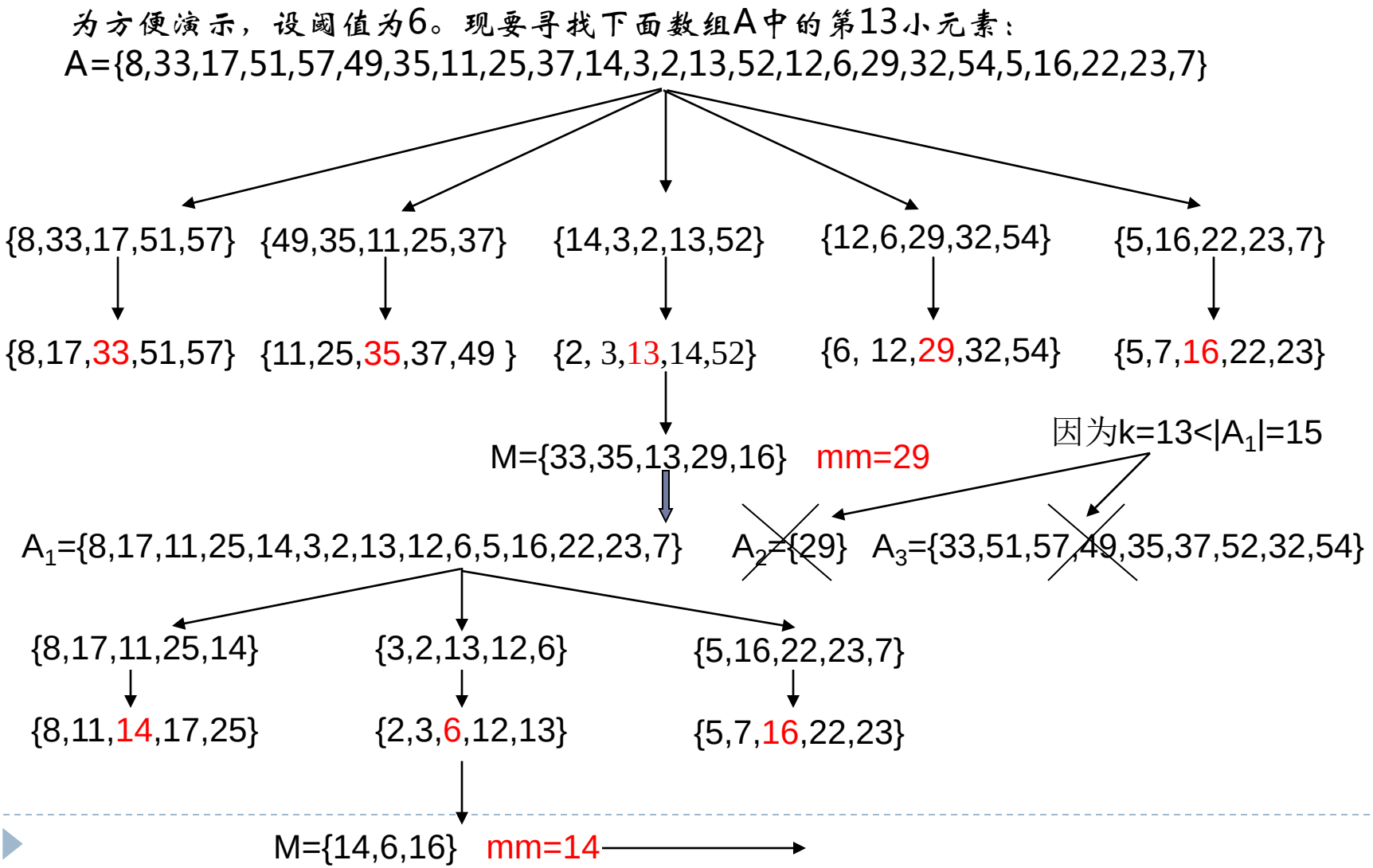
Algorithm: Select(A[low…high], k)

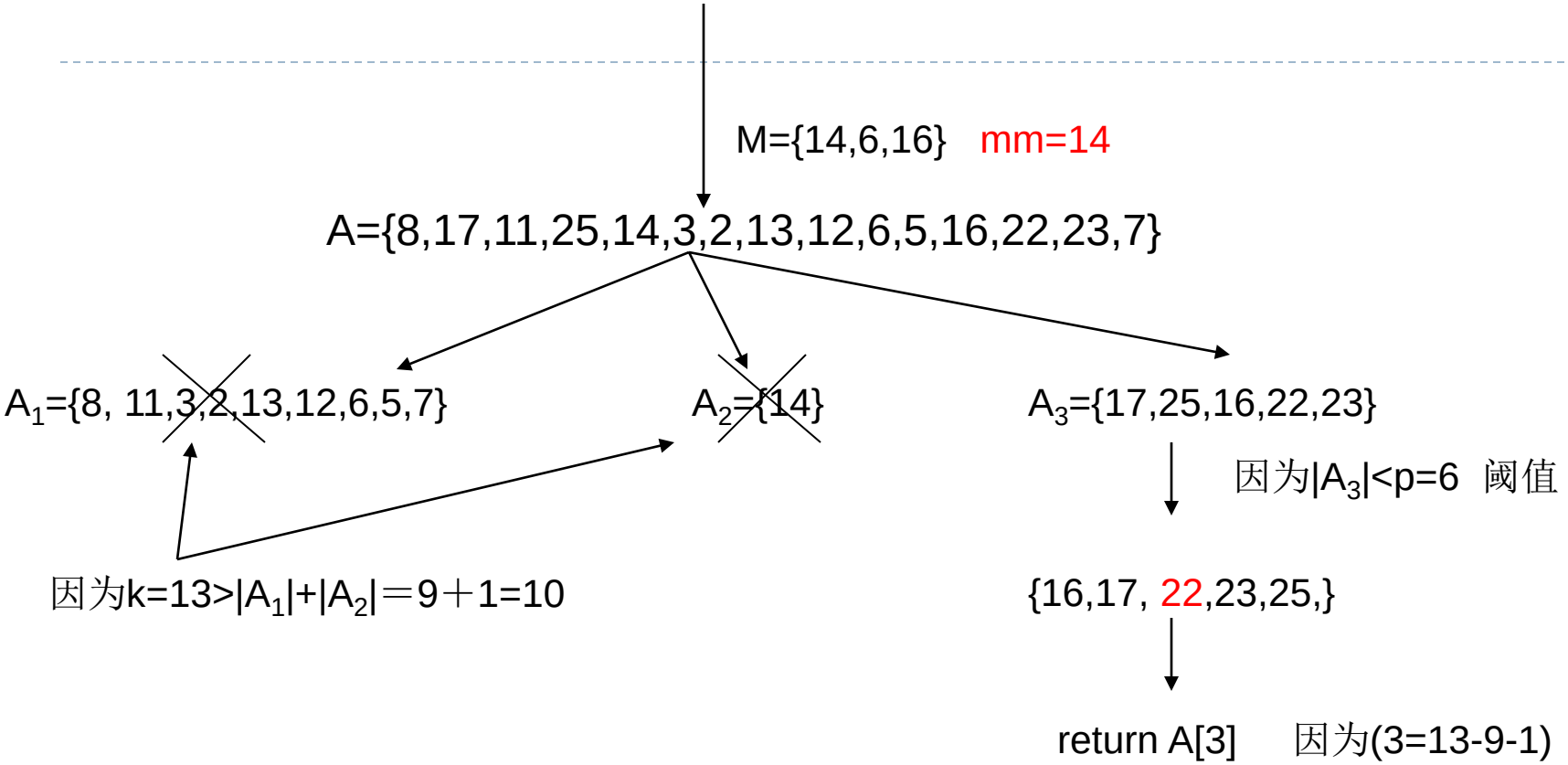
输入：数组A[low,...high]和整数k，$1 \le k \le$ high-low+1

输出：A[low…high]中的第k小元素

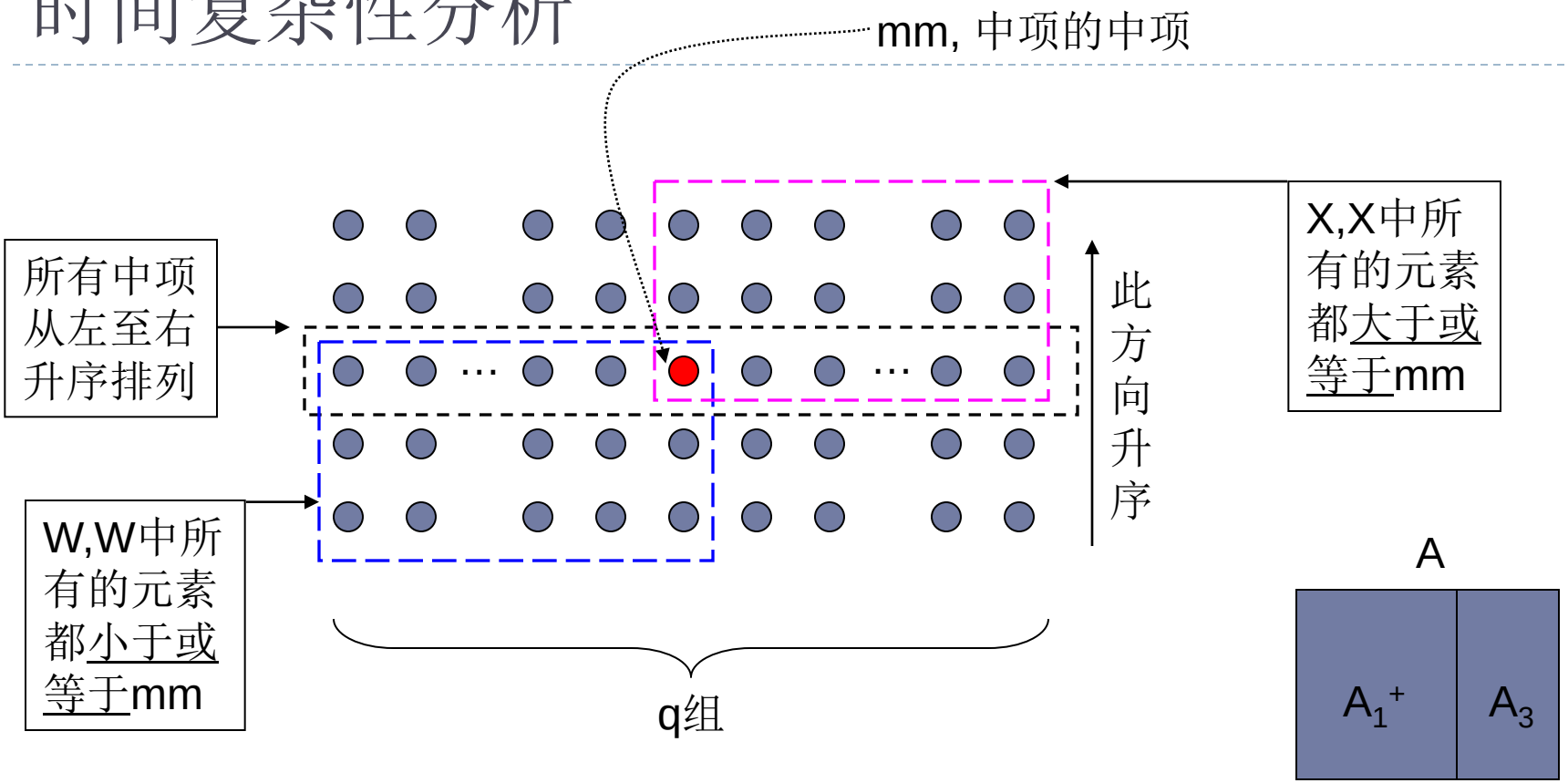
1. n ← high-low+1 //问题的规模 Θ(1)
2. if n <44 then {将 A[low,...high] 排序, return A[low+k-1] // Θ(1)
3. 令q=⌊n/5⌋。将A[low…high]分成q个子数组，每组5个元素。若5不整除n,则排除剩余的元素。 // Θ(n)
4. 对q个子数组分别进行排序，分别找出中项。这些中项组成一个数组M。 // Θ(n)
5. mm ← Select(M[1…q],⌈ q/2 ⌉) //中项的中项, Θ(T⌊n/5⌋)
6. 将A[low…high]分成三组
   A_1＝{a|a<mm} A_2＝{a|a=mm} A_3={a|a>mm}。 //Θ(n)
7. Case //至多T(0.7n+1.2)
   | A_1 | ≥ k : return select(A_1[1…|A_1|],k)
   | A_1 |+| A_2 | ≥ k : return mm
   | A_1 |+| A_2 | < k: return select(A_3[1,|A_3|], k-|A_1|-|A_2|)

一个例子：


为方便演示，设阈值为6。现要寻找下面数组A中的第13小元素：
A={8,33,17,51,57,49,35,11,25,37,14,3,2,13,52,12,6,29,32,54,5,16,22,23,7}
{8,33,17,51,57}
{49,35,11,25,37}
{14,3,2,13,52}
{12,6,29,32,54}
{5,16,22,23,7}
{8,17,33,51,57}
{11,25,35,37,49 }
{2, 3,13,14,52}
{6, 12,29,32,54}
{5,7,16,22,23}
因为k=13<|A1|=15
M={33,35,13,29,16} mm=29
A1={8,17,11,25,14,3,2,13,12,6,5,16,22,23,7}
A2={29}
A3={33,51,57,49,35,37,52,32,54}
{8,17,11,25,14}
{3,2,13,12,6}
{5,16,22,23,7}
{8,11,14,17,25}
{2,3,6,12,13}
{5,7,16,22,23}
M={14,6,16} mm=14

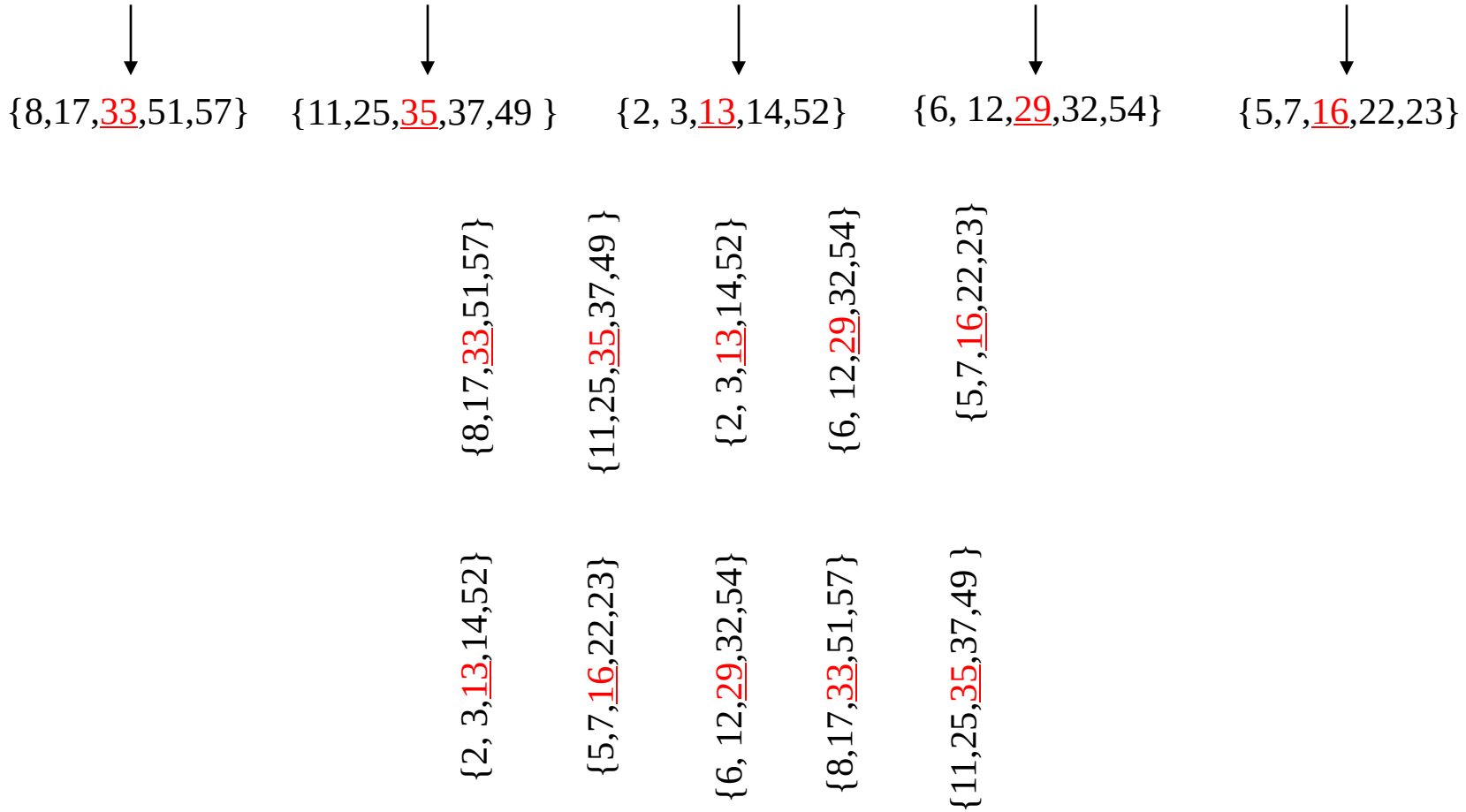

M={14,6,16} mm=14
A={8,17,11,25,14,3,2,13,12,6,5,16,22,23,7}
A1={8, 11,3,2,13,12,6,5,7}
A2={14}
A3={17,25,16,22,23}
因为|A3|<p=6 阈值
因为k=13>|A1|+|A2|＝9＋1=10
{16,17, 22,23,25,}
return A[3] 因为(3=13-9-1)

时间复杂性分析



- A_1^+表示A中小于或等于mm的元素集,A_1是A中严格小于mm的元素集。
- A_3^+表示A中大于或等于mm的元素集,A_3是A中严格大于mm的元素集。
- 因为A_1^+至少与W同样大(为什么？)，所以$|A_1^+| \geq |W| = 3\lceil q/2 \rceil = 3\lceil \lfloor n/5 \rfloor /2 \rceil \geq 3/2\lfloor n/5 \rfloor$，所以，$|A_3| \leq n - 3/2\lfloor n/5 \rfloor \leq n - 3/2\,((n-4)/5) = 0.7n+1.2$
- 由对称性，我们有$|A_1| \leq 0.7n+1.2$

示例:


{8,17,33,51,57}
{11,25,35,37,49 }
{2, 3,13,14,52}
{6, 12,29,32,54}
{5,7,16,22,23}
{8,17,33,51,57}
{11,25,35,37,49 }
{2, 3,13,14,52}
{6, 12,29,32,54}
{5,7,16,22,23}
{2, 3,13,14,52}
{5,7,16,22,23}
{6, 12,29,32,54}
{8,17,33,51,57}
{11,25,35,37,49 }

- 至此，我们为A_1和A_3中的元素个数建立了一个上界：即小于mm的元素个数或是大于mm的元素的个数均不超过0.7n+1.2。
- T(n)表示从n个元素中选择第k小元素所需要耗费的时间。
- Step 1，2耗费时间均为Θ(1)。
- Step 3耗费时间为Θ(n);Step 4耗费时间为Θ(n)。
- Step 5耗费时间为T(⌊n/5⌋)。
- Step 6耗费时间为Θ(n)
- Step 7耗费时间至多为T(0.7n+1.2)。下面设法去掉其中的常数1.2。假设0.7n+1.2 ≤⌊0.75n⌋,那么当0.7n+1.2 ≤0.75n-1，即当n ≥44时， 0.7n+1.2 ≤⌊0.75n⌋成立。此时，Step 7耗费时间之多为T(⌊0.75n⌋).

$$T(n) \le \begin{cases} c & if\ n < 44 \\ T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor) + \Theta(n) & if\ n \ge 44 \end{cases}$$

↓ 定理2.7

$$T(n) = \Theta(n)$$

最接近点对问题

- 给定平面上的点集S，|S|=n。
- 若p∈S,q∈S,p!=q,则(p,q)称为一个点对。
- d(p,q)表示该点对(p,q)中点p和点q之间的欧几里得距离，
- 问d(p,q)的最小值是多少？

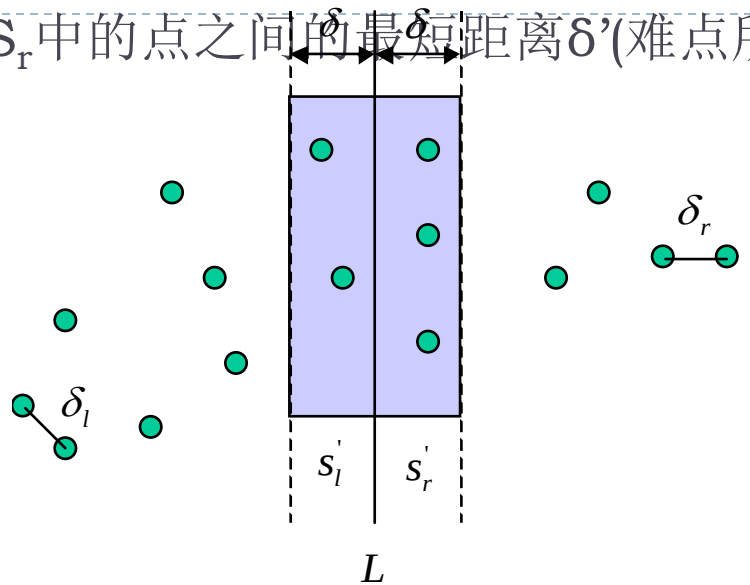
$$\delta = \min_{p \in S, q \in S, p \neq q} d(p,q)$$

- 严格说来，最接近点对可能多于1，为了简单起见，我们找到其中的1对作为问题的解。
- 最直观的方法：将每一点与其它n-1个点的距离算出来，然后找出其中最小的即可。 $T(n)=\Theta(n(n-1)/2)=\Theta(n^2)$

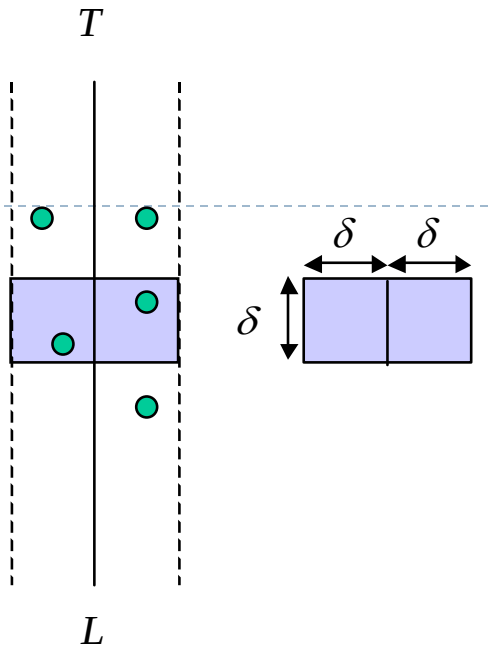
使用分治策略

- 首先，将点集S中的点按照x坐标进行排序。
- 其次，使用一根垂直于x轴的直线L将S分割为两个子集S_l和S_r，使得$|S_l| = \lfloor |S|/2 \rfloor$ $|S_r| = \lceil |S|/2 \rceil$。$S_l$中的点都落在直线L的左边或是L上；$S_r$中的点都落在直线L的右边或是L上。
- 计算S_l中的最小距离δ_l，S_r中的最小距离δ_r(递归)； 计算S_l中的点与S_r中的点之间的最短距离δ'(难点所在)。
- 最接近点对的距离为$\min\{\delta_l, \delta_r, \delta'\}$。

计算S_l中的点与S_r中的点之间的最短距离δ’(难点所在)



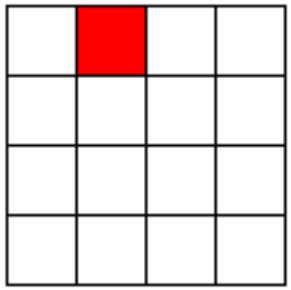
- 直接的方法：求解S_l中每个点与S_r中每个点之间距离，然后找出最小值。然而，这种方法需要的时间$T(n)=\Theta(n^2)$。
- 我们设$\delta = \min\{\delta_l,\delta_r\}$，如果最近邻点对是由 S_l中的某个点p和S_r中的某个点q所组成，那么，p,q至L的水平距离均不会超过δ。
- 用S_l' 表示S_l中与L水平距离小于等于δ的点集，用S_r' 表示S_r中与L水平距离小于等于δ的点集。也就是有$p \in S_l'$, $q \in S_r'$
- 然而，计算S_l' 和S_r' 中点对的最小值在最坏情形下时间复杂性仍旧为$T(n)=\Theta(n^2)$ （当$S_l' = S_l$和$S_r' = S_r$时）。

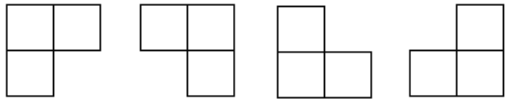


- 用T表示两个垂直带之间的点的集合。
- 进一步分析，可以发现：如果最近邻点对是由 S_l中的某个点p和S_r中的某个点q所组成，那么，p,q之间的垂直距离不会超过δ。
- 在一个长为2δ,宽为δ的矩形内，任意两点之间的距离不超过δ，那么该矩形内至多容纳8个点(其中4个属于S_l',4个属于S_r')。所以T中的每个点至多只需要和T中的7个点计算距离(O(7n))，而不需要计算和T中所有的其它点之间的距离(O(n^2))。
- 至此，问题得解。课后自行阅读CLOSESTPAIR算法，及时间复杂性分析。

棋盘覆盖问题

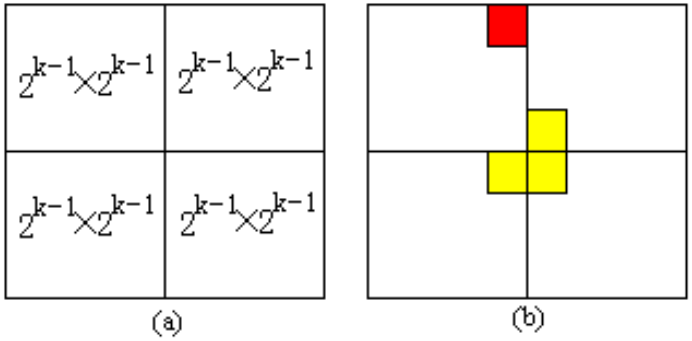
在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为一**特殊方格**(红色表示)，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。





分析

当k>0时，可以将$2^k \times 2^k$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	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

一个实例

算法描述

```
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);
      }
  }
```

tr: 棋盘左上角方格的行号
tc: 棋盘左上角方格的列号
dr: 特殊方格所在的行号
dc: 特殊方格所在的列号
size: 棋盘大小

$$T(k) = 4T(k-1) + O(1)$$
$$= \Theta(4^k)$$

小结

- **分治法的适用条件**
  - 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同类问题；
  - 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(问题规模较小的时候)，但一般用动态规划更为合适。

课堂测试

用一种组合论证，证明如果 n 是 2 的幂，则 $n^2 \equiv 1 \pmod 3$

课堂测试

题4-5

4-5 （芯片检测） Diogenes 教授有 n 片可能完全一样的集成电路芯片，原理上可以用来相互检测。教授的测试夹具同时只能容纳两块芯片。当夹具装载上时，每块芯片都检测另一块，并报告它是好是坏。一块好的芯片总能准确报告另一块芯片的好坏，但教授不能信任坏芯片报告的结果。因此，4 种可能的测试结果如下：

芯片 A 的结果	芯片 B 的结果	结　论
B 是好的	A 是好的	两片都是好的，或都是坏的
B 是好的	A 是坏的	至少一块是坏的
B 是坏的	A 是好的	至少一块是坏的
B 是坏的	A 是坏的	至少一块是坏的

a. 证明：如果超过 $n/2$ 块芯片是坏的，使用任何基于这种逐对检测操作的策略，教授都不能确定哪些芯片是好的。假定坏芯片可以合谋欺骗教授。

b. 考虑从 n 块芯片中寻找一块好芯片的问题，假定超过 $n/2$ 块芯片是好的。证明：进行 $\lfloor n/2 \rfloor$ 次逐对检测足以将问题规模减半。

c. 假定超过 $n/2$ 块芯片是好的，证明：可以用 $\Theta(n)$ 次逐对检测找出好的芯片。给出描述检测次数的递归式，并求解它。