算法设计与分析

分治策略

主要内容

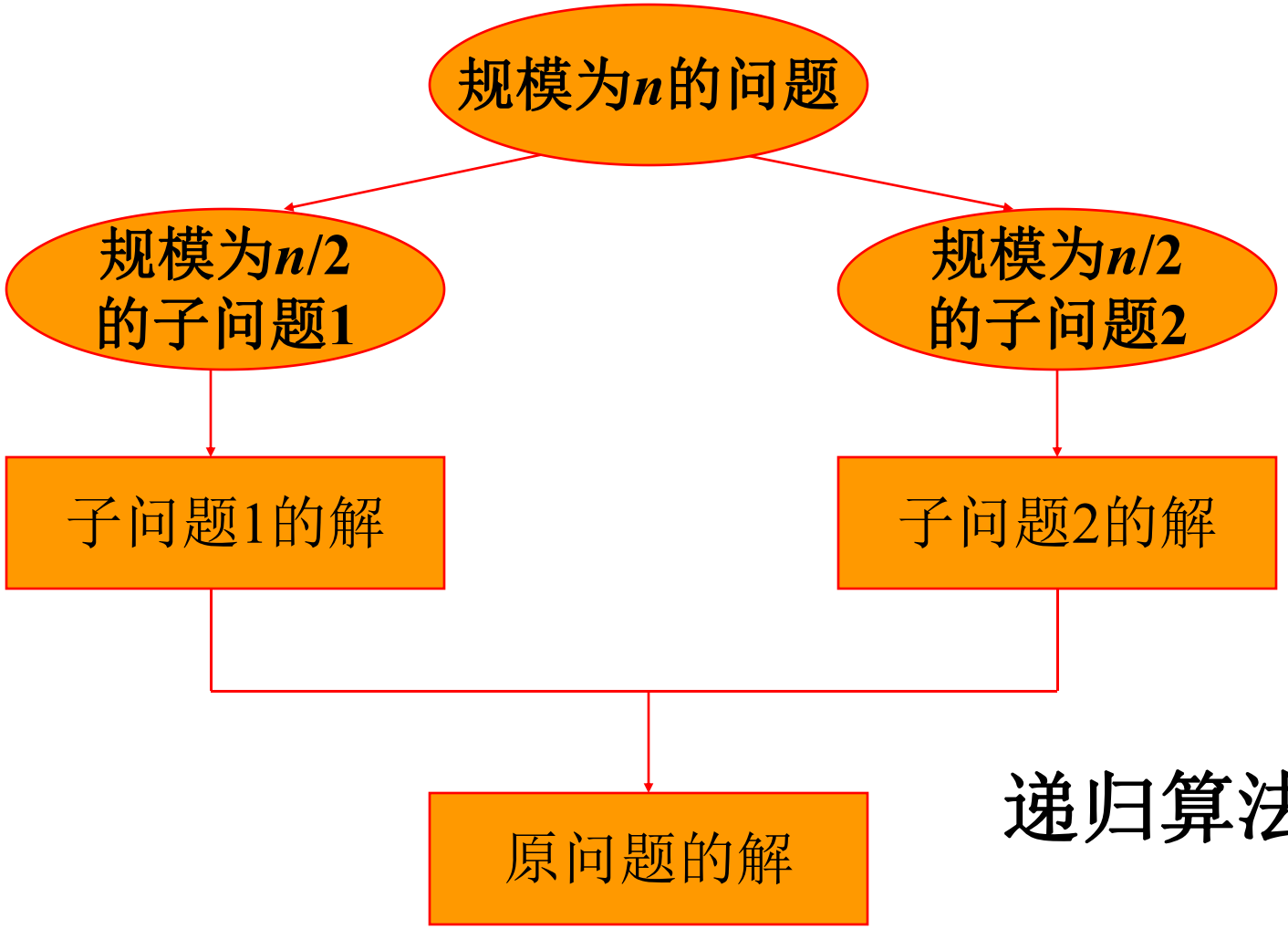
- 方法
  - 分治策略
  - 分治法效率分析——迭代法（递归树法）
  - 分治法效率分析——主定理方法
- 问题
  - 最近点对问题
  - 凸包问题
  - 最大子数组问题
  - 矩阵乘法的Strassen算法

分治策略

- 将一个问题分解为与原问题相似但规模更小的若干子问题，递归地解这些子问题，然后将这些子问题的解结合起来构成原问题的解。这种方法在每层递归上均包括三个步骤

  - **Divide（分解）：将问题划分为若干个子问题**
  - **Conquer（求解）：递归地解这些子问题；若子问题Size足够小，则直接解决之**
  - **Combine（组合）：将子问题的解结合成原问题的解**

分治法



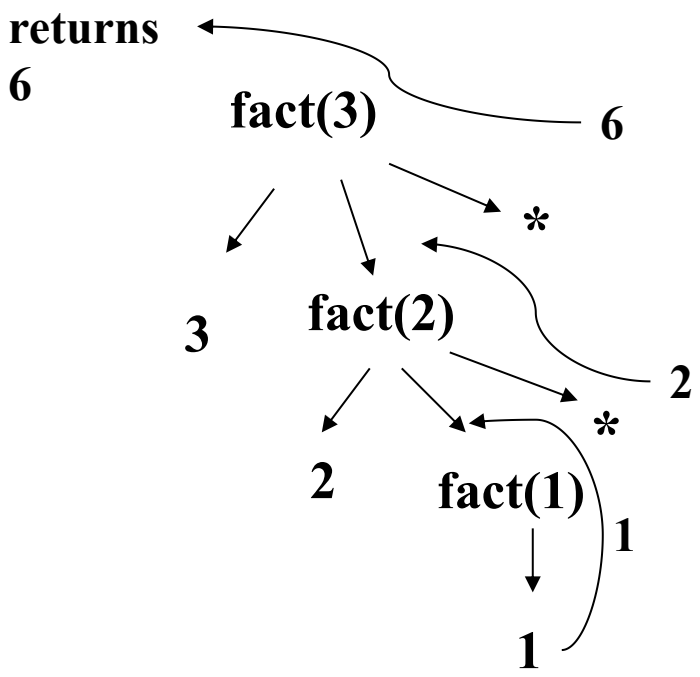
递归算法！

递归算法

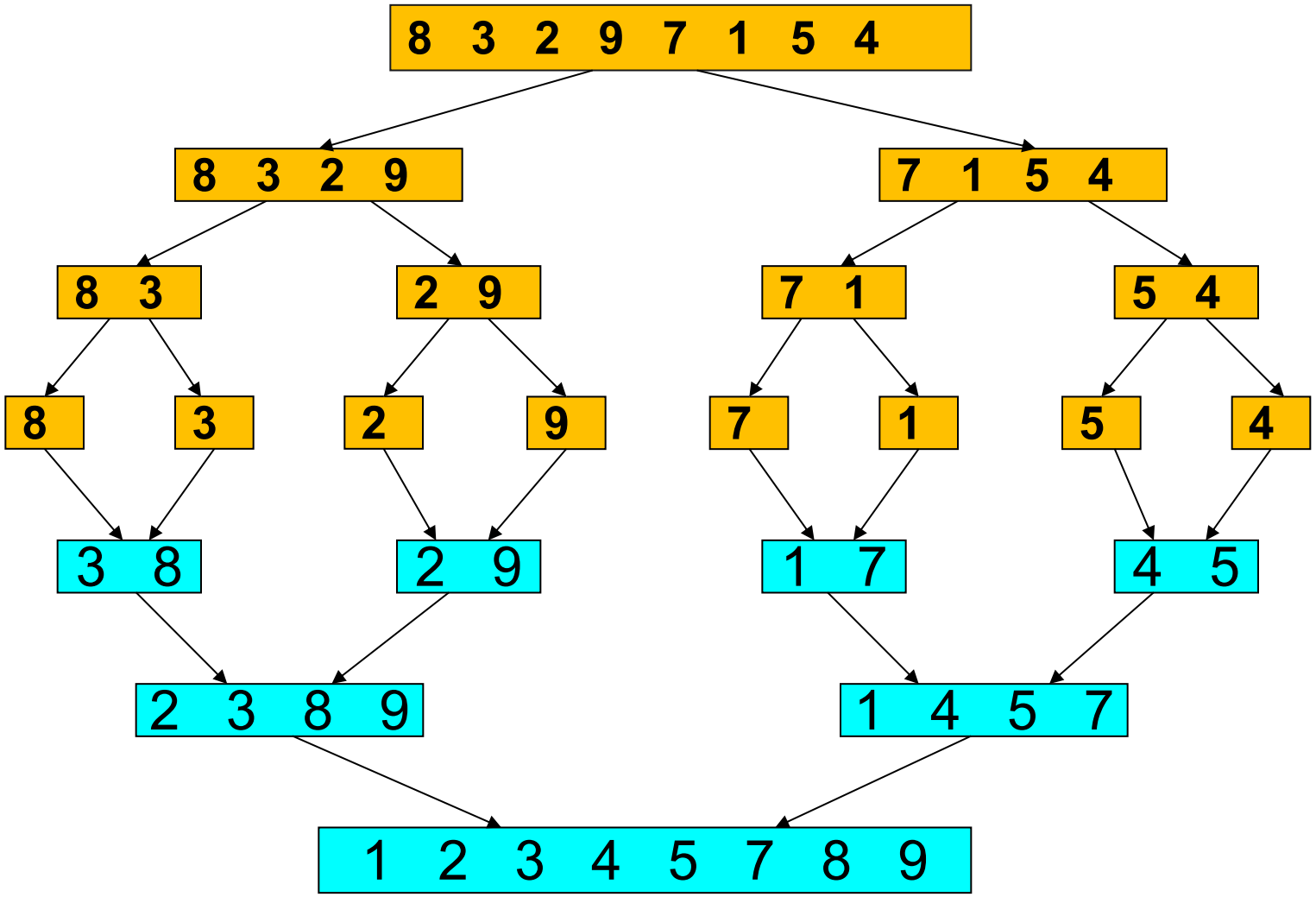
- 一个递归算法通常包含递归的调用该算法本身，传入较小的参数。
- 递归算法的中止条件：
  - 处理基本情况，这些情况不可以有任何递归调用。

例子：求n！

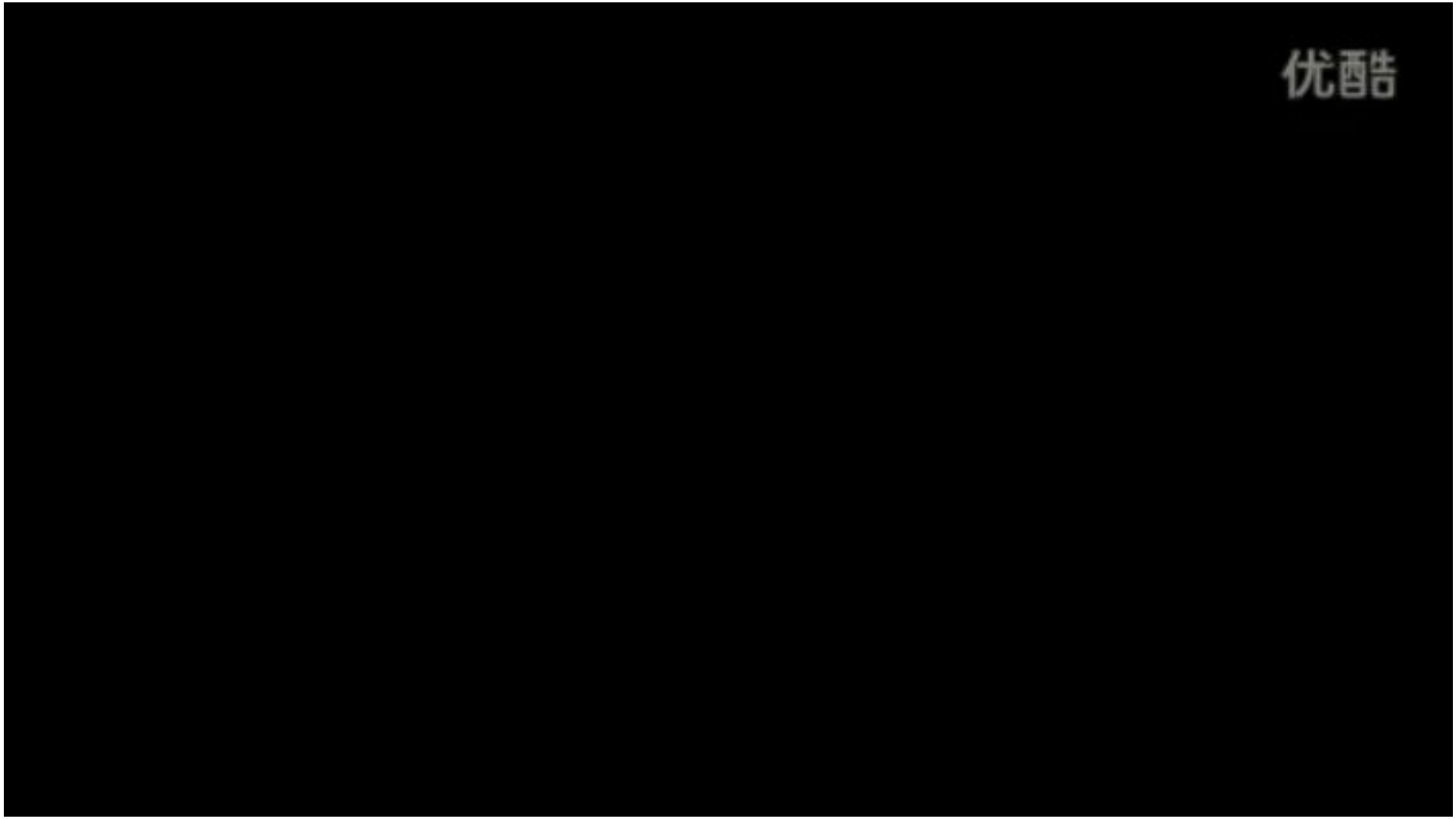
```
int fact(int n) {
    if (n<=1) return 1;
   else return n*fact(n-1);
}
```



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 7 8 9

视频欣赏



归并排序

- 输入数组 $A[p \,..\, r]$

$\textsc{Merge-Sort}(A, p, r)$

if $p < r$

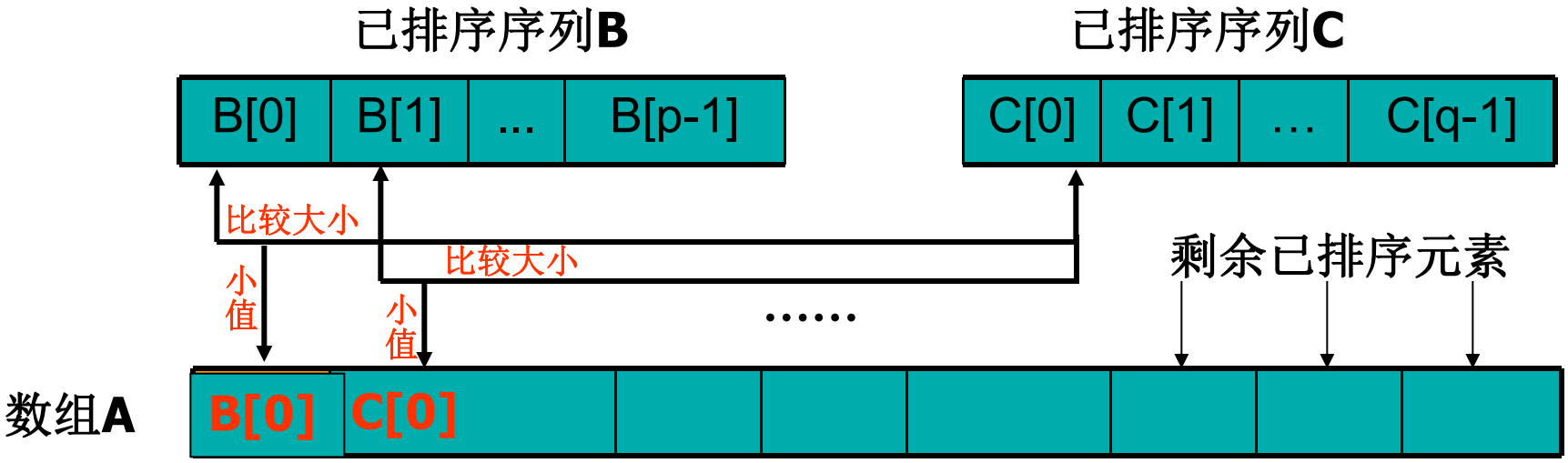
$q = \lfloor (p + r)/2 \rfloor$

$\textsc{Merge-Sort}(A, p, q)$

$\textsc{Merge-Sort}(A, q + 1, r)$

$\textsc{Merge}(A, p, q, r)$

合并函数MERGE的实现



分治算法的效率分析

- 用递归式分析分治算法的运行时间。
- 一个递归式是一个函数，它由一个或多个基本情况（base case），它自身，以及小参数组成。
- 递归式的解可以用来近似算法的运行时间。

例子：求n！

```
int fact(int n) {
    if (n<=1) return 1;
    else return n*fact(n-1);
}
```

递归关系式：

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

递归式求解方法1——迭代法(1)

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

当n>1时

T(n) = T(n-1)+1

=T(n-2)+1+1

=…

=T(1)+1+…+1

=T(1)+n-1

= n

$$\therefore T(n)\in\Theta(n)$$

归并排序

- 输入数组 $A[p \,..\, r]$

$\textsc{Merge-Sort}(A, p, r)$

if $p < r$

$q = \lfloor (p + r)/2 \rfloor$

$\textsc{Merge-Sort}(A, p, q)$

$\textsc{Merge-Sort}(A, q + 1, r)$

$\textsc{Merge}(A, p, q, r)$

- 递归式:

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

递归式求解方法1——迭代法

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

$if \quad n=2^k$

$$\begin{aligned}T(2^k)&=2T(2^{k-1})+2^k\\&=2\left(2T(2^{k-2})+2^{k-1}\right)+2^k\\&=2^2T(2^{k-2})+2^k+2^k\\&=\mathrm{L}\\&=2^kT(2^{k-k})+2^k+\mathrm{L}\ \ +2^k\\&=2^k+k2^k\end{aligned}$$

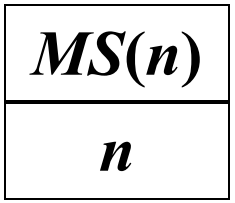
$$\therefore T(n)\in\Theta\left(n\log n\right)$$

递归式求解方法2——递归树法

- 递归树给出了递归算法中各个过程运行时间的估计。
- 递归树每次在深度上扩展一层。
- 递归式T(n) = kT(n / m) + f(n)中每次递归调用用一个结点表示，结点包含非递归操作次数f(n)。

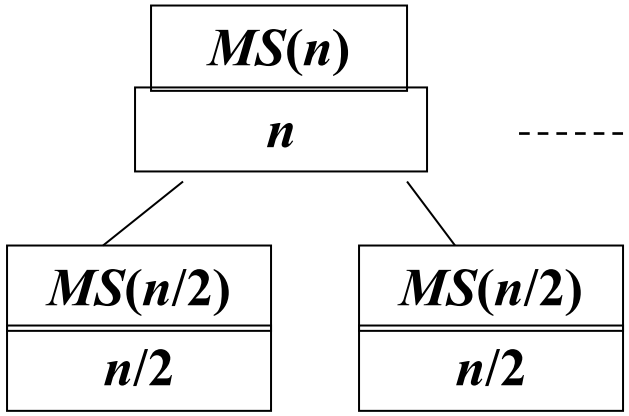
  例如：T(n) = 2T(n/2) + n, 非递归部分为 n， n就是节点下面的值 。
- 每个节点的分支数为k
- 每层的右侧标出当前层中所有节点的和。
- 将所有层总的操作次数相加。

归并排序 (*MS*) 的递归树



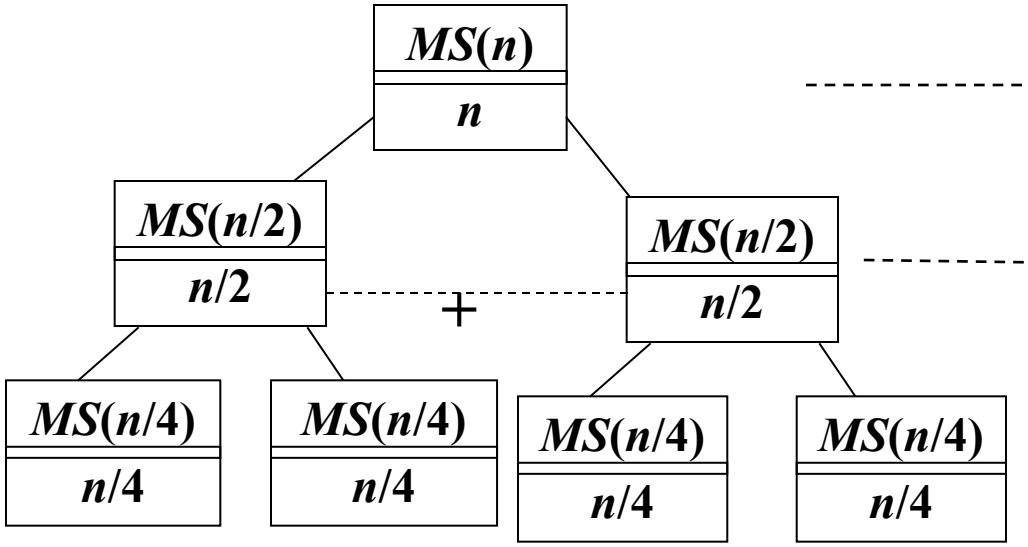
- 初始的，递归树只有一个结点，包含调用 $MS(n)$, 最坏情况下的合并代价是 n.
- 当展开计算，该节点变成一颗子树:
  - 子树的根结点包含调用 $MS(n)$, 和非递归的操作 cn.
  - 两个孩子结点各包含一个递归调用 $MS(n/2)$, 最坏情况下的合并代价是 $c(n/2)$。

归并排序第一层展开



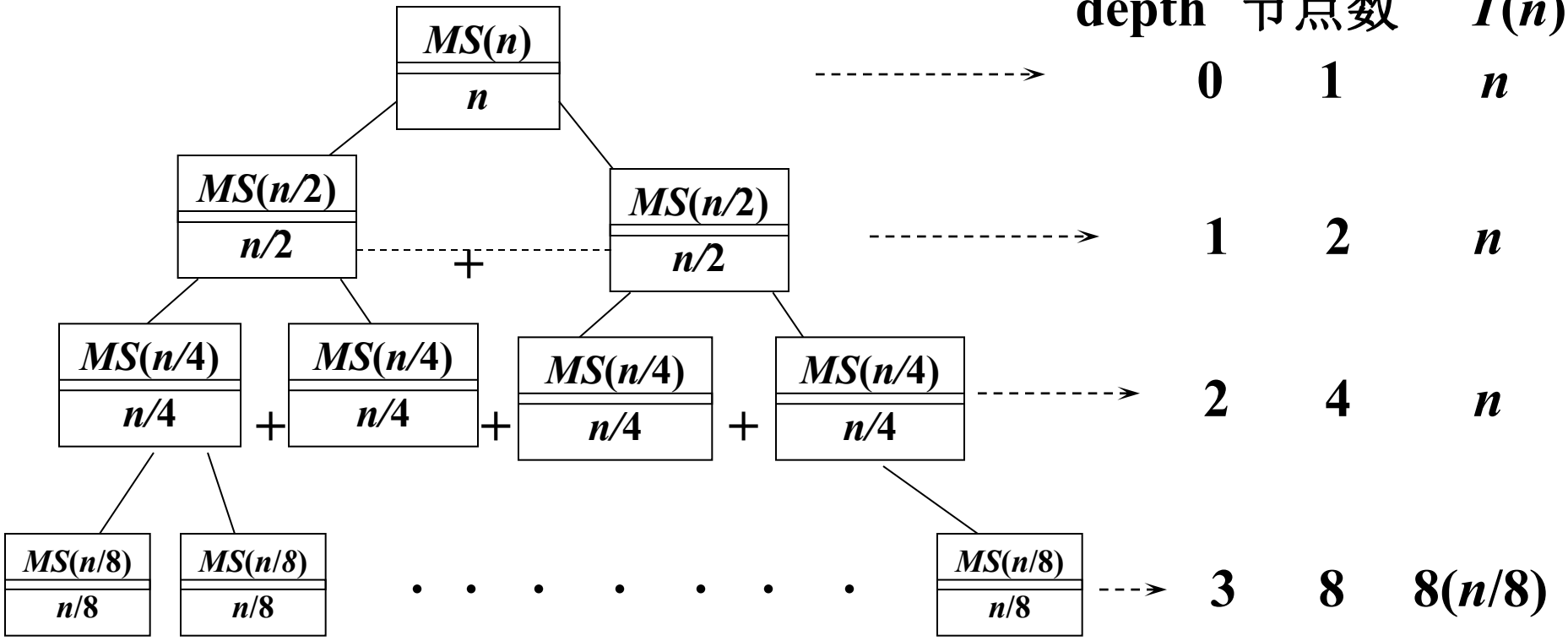
depth	节点数	$T(n)$
0	1	n
1	2	$2(n/2)$

第二层展开



depth	节点数	$T(n)$
0	1	n
1	2	n
2	4	$T(n/4)$

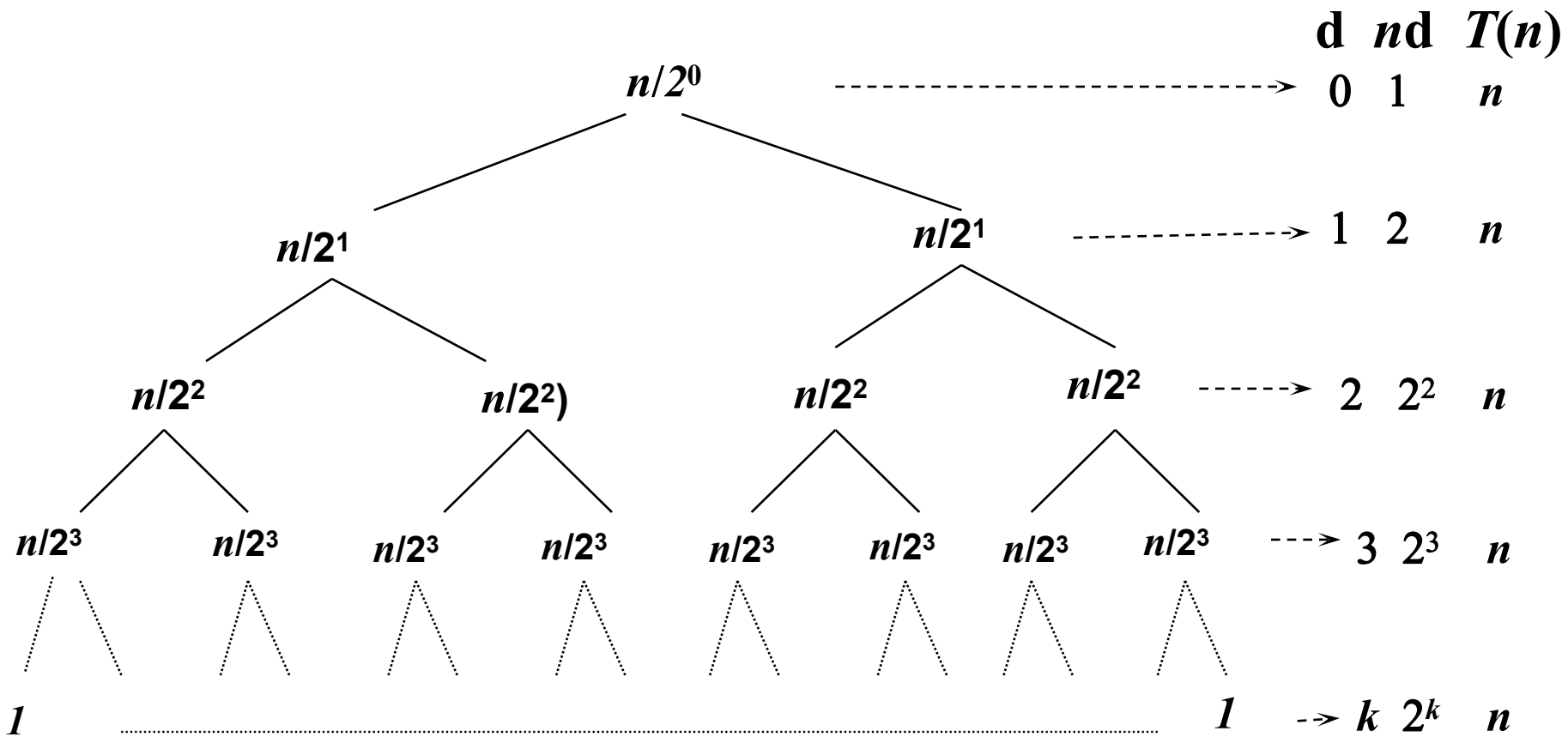
第三层展开


MS(n)
n
MS(n/2)
n/2
+
MS(n/2)
n/2
MS(n/4)
n/4
+
MS(n/4)
n/4
+
MS(n/4)
n/4
+
MS(n/4)
n/4
MS(n/8)
n/8
MS(n/8)
n/8
MS(n/8)
n/8
depth 节点数 T(n)
0 1 n
1 2 n
2 4 n
3 8 8(n/8)

中止展开

- 简化 $n = 2^k$ ➔ $\lg n = k$.
- 当一个结点调用 $MS(n/2^k)$:
  - 归并排序输入的规模为 $n/2^k = 1$.
  - 这种情况下展开中止, 该节点为叶子结点，代价是 $\Theta(1)$

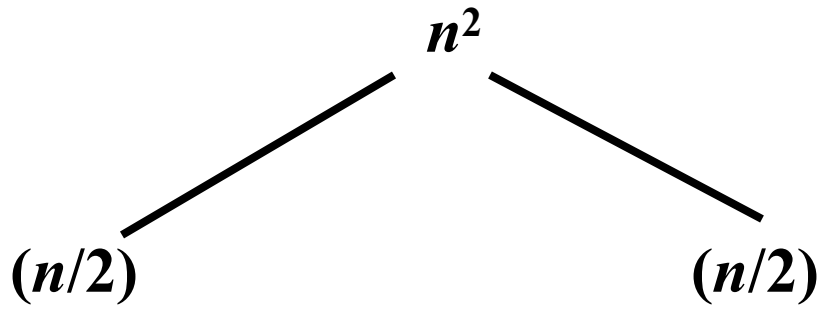
完整的递归树



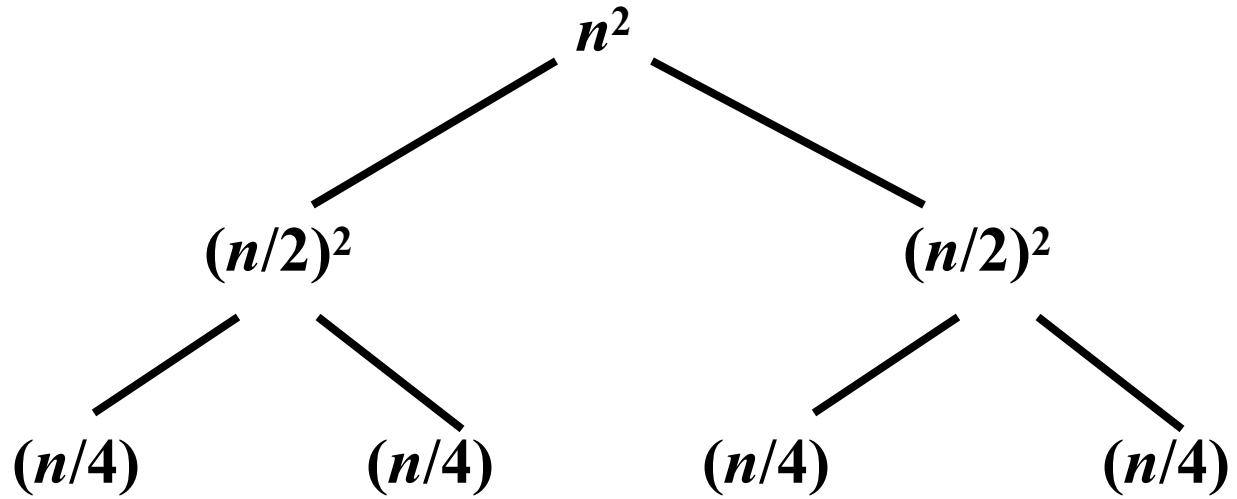
$$T(n) = (k+1)(n) = (\lg n + 1)(n) = \Theta(n \lg n)$$

举例: $T(n) = 2T(n/2) + n^2$

第一层展开:

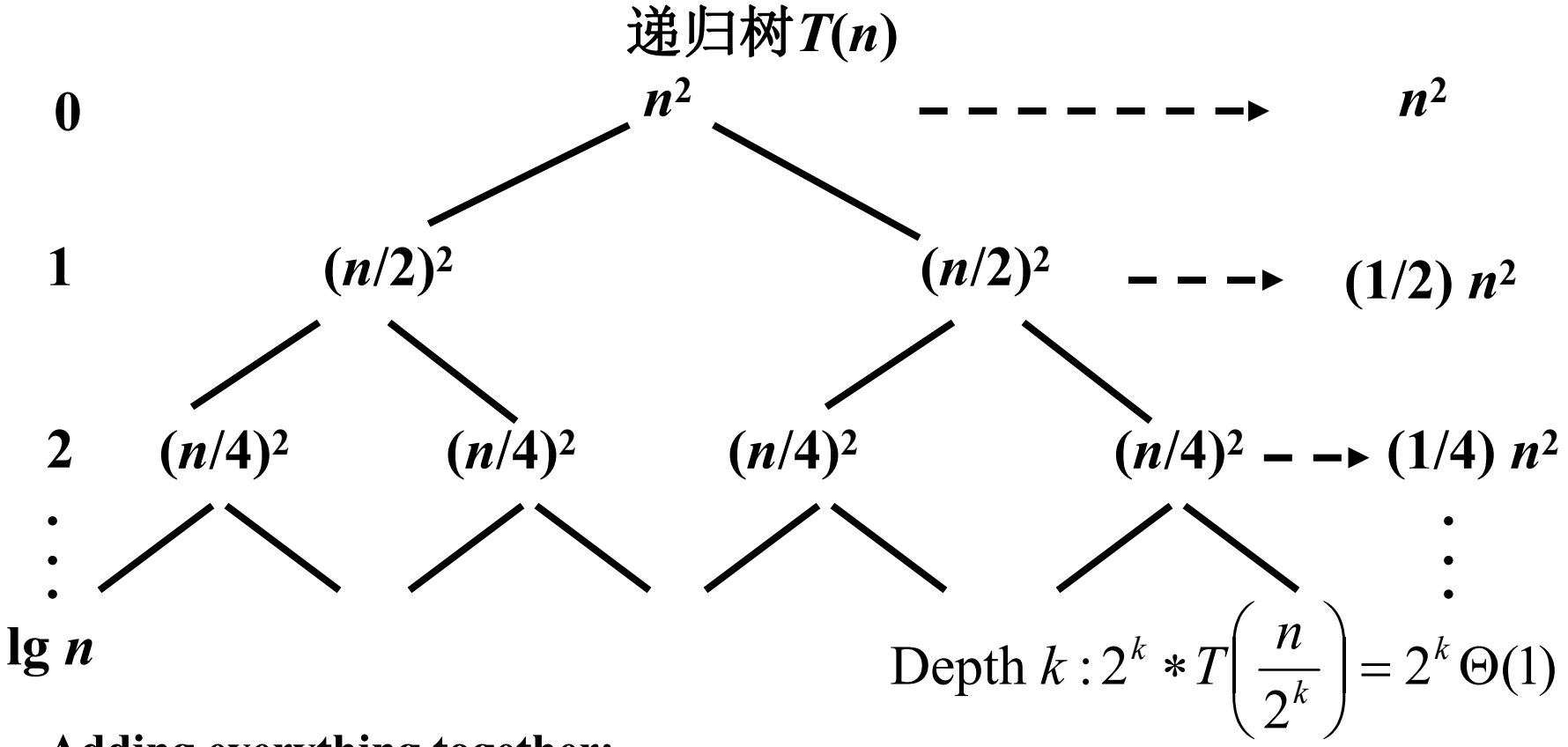


第二层展开:



Let $n = 2^k$. Then $k = \lg n$, $n/2^k = 1$, $(n/ 2^k) = (1) = \Theta(1)$.

举例: $T(n) = 2T(n/2) + n^2$

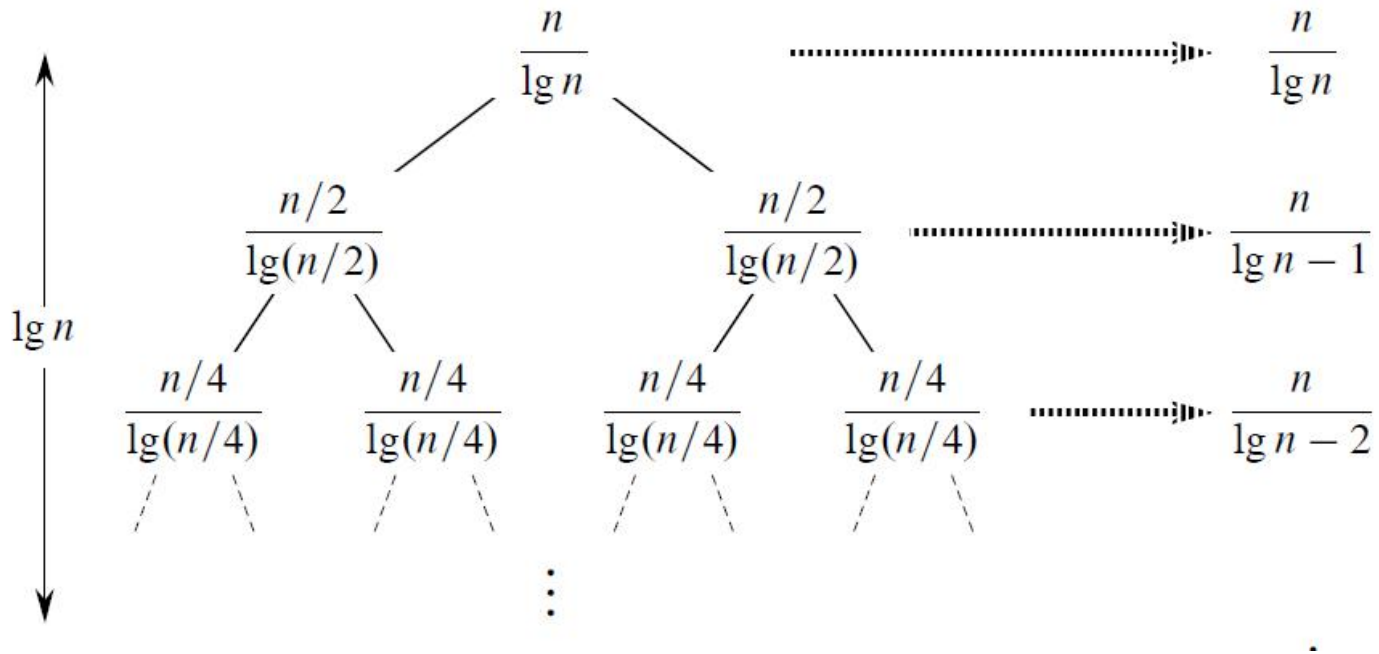


Adding everything together:

$$\sum_{i=0}^{k-1} \frac{1}{2^i} n^2 + 2^k \Theta(1) = n^2 \left(\frac{1-(1/2)^k}{1-1/2} \right) + n\Theta(1) = 2n^2(1-1/n) + n\Theta(1) = \Theta(n^2)$$

举例 4 (续)

- 用递归树方法解 $T(n) = 2T(n/2) + n / \lg n$



- 在深度 i , 有 2^i 结点, 每个是

$$(n/2^i) / \lg (n/2^i) = (n/2^i) / (\lg n - i)$$

➔ 深度 i 的代价是 $2^i (n/2^i) / (\lg n - i) = n / (\lg n - i)$

举例 4 (续)

- 把每一层的代价加起来

$$\sum_{i=0}^{\lg n-1}\frac{n}{\lg n-i}=\frac{n}{\lg n}+\frac{n}{\lg n-1}+\cdots\cdots+\frac{n}{2}+\frac{n}{1}=\sum_{i=1}^{\lg n}\frac{n}{i}$$

$$=n\sum_{i=1}^{\lg n}\frac{1}{i}=n(\lg\lg n+O(1))=\Theta(n\lg\lg n)$$

➔ $T(n)=\Theta(n\lg\lg n)$

调和级数:

$$\sum_{i=1}^{n}\frac{1}{i}=\lg n+O(1)$$

递归式求解方法2——主定理法（1）

- 该方法可解如下形式的递归式

$$T(n) = aT(n/b) + f(n)$$

其中 $a \geq 1$ 和 $b > 1$ 是两个常数, $f(n)$ 是一个渐进非负函数（当n趋于无穷时， $f(n)$ 是非负的）。

  - 如果 n/b 不是整数, 取整 n/b :

$$\lfloor n/b \rfloor \text{ or } \lceil n/b \rceil.$$

- 主方法可解包含三种类型 $f(n)$ 的递归式 $T(n)$ 。

递归式求解方法2——主定理法（2）

$$T(n)=\begin{cases}\Theta(n^{\log_b a}) & \text{if } f(n)\in O(n^{\log_b a-\varepsilon}),\ \varepsilon>0 \quad (1)\\ \Theta(n^{\log_b a}\lg^{k+1} n) & \text{if } f(n)\in \Theta(n^{\log_b a}\lg^{k} n),\ k\ge 0 \quad (2)\\ \Theta(f(n)) & \text{if } f(n)\in \Omega(n^{\log_b a+\varepsilon}),\ \varepsilon>0 \text{ and} \quad (3)\end{cases}$$

$af(n/b)\le cf(n)$ for some

$c<1$ and all sufficiently large n

正则条件

$$T(n)=8T\left(\frac{n}{2}\right)+1000n^2$$

$$T(n)=2T\left(\frac{n}{2}\right)+10n$$

$$T(n)=2T\left(\frac{n}{2}\right)+n^2$$

理解主定理（1）

关键是看 $f(n)$ 和 $n^{\log_b a}$ 谁比较大。

- **Case 1 成立，如果 $n^{\log_b a}$ 较大 ➔ $T(n) = \Theta(n^{\log_b a})$.**
  - “较大” 指*多项式意义上的大*, 大一个因子 n^{ε}, for some $\varepsilon > 0$.

例如：

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

$$a = 8,\ b = 2,\ f(n) = 1000n^2,\ \text{so}$$

$$f(n) \in O(n^c),\ \text{where}\ c = 2$$

$$\log_b a = \log_2 8 = 3 > c.$$

$$T(n) \in \Theta\left(n^{\log_b a}\right) = \Theta\left(n^3\right)$$

理解主定理（2）

- **Case 2 (当 $k = 0$) 成立，如果 $f(n)$ 和 $n^{\log_b a}$ 大小相当。**
  - 这种情况，乘以一个对数因子 ➔ $T(n) = \Theta(f(n)\lg n)$.
  - 一般来说, 当 $f(n)$ 和 $n^{\log_b a}\lg^k n$ 大小相当

    ➔ $T(n) = \Theta(f(n)\lg^{k+1} n)$.

- **Case 2的特殊情况: $k = 0$**

$$T(n) = \Theta(n^{\log_b a}\lg n) \quad \text{if } f(n) \in \Theta(n^{\log_b a})$$

理解主定理（3）

- 例：

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ c = 1,\ f(n) = 10n$$

$$f(n) = \Theta\left(n^c \log^k n\right) \text{ where } c = 1, k = 0$$

Next, we see if we satisfy the case 2 condition:

$$\log_b a = \log_2 2 = 1, \text{ and therefore, yes, } c = \log_b a$$

So it follows from the second case of the master theorem:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right) = \Theta\left(n^1 \log^1 n\right) = \Theta\left(n \log n\right)$$

理解主定理（4）

- **Case 3 成立，如果 $f(n)$ is 较大 ➔ $T(n) = \Theta(f(n))$.**
  - **Case 3 要满足正则条件（*regularity condition*）. 正则条件对于 $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a + \varepsilon})$ 存在 $\varepsilon > 0$ 总是成立的。**
  - **例** $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ f(n) = n^2$$

$$f(n) = \Omega\left(n^c\right),\ \text{where}\ c = 2$$

Next, we see if we satisfy the case 3 condition:

$$\log_b a = \log_2 2 = 1,\ \text{and therefore, yes,}\ c > \log_b a$$

The regularity condition also holds:

$$2\left(\frac{n^2}{4}\right) \leq kn^2,\ \text{choosing}\ k = 1/2$$

So it follows from the third case of the master theorem:

$$T(n) = \Theta(f(n)) = \Theta\left(n^2\right).$$

正则条件 (1)

- 正则条件是什么？
  - 1. 在递归式$T(n) = aT(n/b) + f(n)$中，$f(n)$ 可以直观的被解释为把一个规模为n的问题分解成a个规模为n/b的子问题和合并a个子问题的解的代价
  - 2. $af(n/b)$ 可以被解释为把a个规模为n/b的子问题分解成a^2 个规模为n/b^2 的子问题和合并a^2 个子问题解的代价。
  - 条件 $af(n/b) \le cf(n)$, for $c < 1$ 和足够大的 n, 可以被解释为上述第一点的代价是上述第二点代价的准确界。

    ➔ 当一个问题被分解成越来越小的子问题，分解和合并的代价变得越来越小。

主定理

主定理另一种形式:

$$T(n)=\begin{cases}\Theta(n^{\log_b a}), & if\ \dfrac{f(n)}{n^{\log_b a}}\in O(n^{-\varepsilon}),\varepsilon>0 \quad (1)\\ \Theta(n^{\log_b a}\lg^{k+1} n), & if\ \dfrac{f(n)}{n^{\log_b a}}\in \Theta(\lg^{k} n),k\geq 0 \quad (2)\\ \Theta(f(n)), & if\ \dfrac{f(n)}{n^{\log_b a}}\in \Omega(n^{\varepsilon}), \quad (3)\\ & \varepsilon>0,\\ & af(n/b)\leq cf(n)\ \text{for some c}<1\\ & \text{and all sufficiently large } n\end{cases}$$

举例 1

- 递归式 $T(n) = 5T(n/2) + n^2$.

  其中 $a = 5$ ， $b = 2$.

  因为 $\log_2 5 > \log_2 4$, $\varepsilon = \log_2 5 - \log_2 4 > 0$.

  因为 $f(n) = n^2 = n^{\log_2 5 - \varepsilon} \in O(n^{\log_2 5 - \varepsilon})$,

  根据 主定理 Case 1

  ➔ $T(n) = \Theta(n^{\log_2 5})$.

举例 2

- 递归式 $T(n) = 27T(n/3) + n^3 \lg n$

  其中 $a = 27$ ， $b = 3$.

  因为 $n^{\log_3 27} = n^3, f(n) = n^3 \lg n = n^{\log_3 27} \lg n$.

  根据主定理 Case 2 中 $k = 1$

  ➔ $T(n) = \Theta(n^3 \lg^2 n)$.

举例 3

- 递归式 $T(n) = 5T(n/2) + n^3$.

  其中 $a = 5$ ， $b = 2$.

  因为 $3 = \log_2 8 > \log_2 5$, $\varepsilon = \log_2 8 - \log_2 5 > 0$.

  ➔ $f(n) = n^3 = n^{\log_2 5 + \varepsilon} \in \Omega(n^{\log_2 5 + \varepsilon})$,

  $af(n/b) = 5f(n/2) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$

  根据主定理 Case 3

  ➔ $T(n) = \Theta(n^3)$.

举例 4

- 递归式 $T(n) = 2T(n/2) + n / \lg n$

  其中 $a = 2$ ， $b = 2$ ➔ $n^{\log_2 2} = n$.

  - $f(n) = n / \lg n$ is not in $O(n^{\log_b a - \varepsilon}) = O(n^{1-\varepsilon})$ for any $\varepsilon > 0$.

    尽管 $n^{\log_b a} > f(n)$, 但不是多项式意义上的大

    ➔ 主定理 Case 1 不适用。
  - $f(n) = n / \lg n$ is not in $\Theta(n^{\log_b a} \lg^k n)$ for any $k \geq 0$

    ➔ 主定理 Case 2 不适用。
  - $f(n) = n / \lg n$ is not in $\Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{1+\varepsilon})$ for any $\varepsilon > 0$.

    ➔ 主定理 Case 3 不适用。

  ➔ $T(n)$ 不能用主定理解。

改变变量

- 有时代数操作可以把一个未知的递归式转换成已知可解的递归式。

举例: 递归式 $T(n) = 2T(\sqrt{n}) + \lg n$.

变量替换 $m = \lg n$, we have $n = 2^m$, and

$$T(2^m) = 2T(2^{m/2}) + m.$$

引入 $S(m) = T(2^m)$, 得出新的递归式:

$$S(m) = 2S(m/2) + m.$$

解 $S(m)$: $\in \Theta(m \lg m)$

➔ $T(n) = T(2^m) = S(m) \in \Theta(m \lg m) = \Theta(\lg n \lg\lg n)$.

最大子数组问题

问题:

- *输入*: 数值数组 $A[1 .. n]$
  - 假设数组中存在负数
  - 如果数组中全是非负数，该问题很简单。

- *输出*: 数组下标 i 和 j 使得子数组 $A[i .. j]$为A[1 .. n]的和最大的非空连续子数组。

最大子数组问题应用

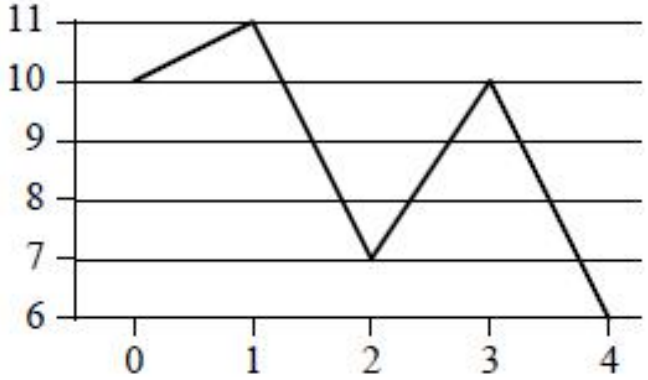
- 考虑下面的情景:
  - 一支股票连续n天的交易价格。
  - 什么时间该买入? 什么时间该卖出?
- 如何将这个问题转换成最大子数组问题?

定义: $A[i]$ = (第i天的价格) – (第$i-1$天的价格)

- 如果最大子数组是$A[i..j]$
  - 第 i 天买入。
  - 第 j 天后卖出。

最大子数组问题应用

- 一支股票连续n天的交易价格:



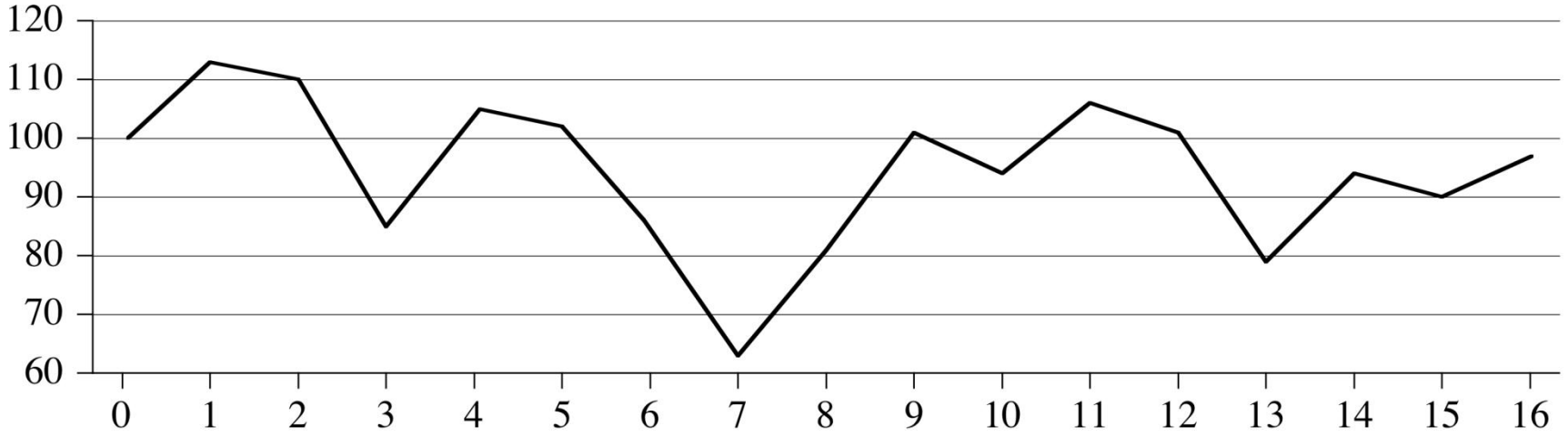
Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

最后一行是 *A*.

- 最大子数组是 *A*[3 .. 3].

最大子数组问题应用

- 一支股票连续n天的交易价格:



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- 最大子数组是 $A[8 .. 11]$.

蛮力法

蛮力法:

- 首先找出所有可能的连续子数组
- 子数组的个数?

$$\binom{n}{2} = \Theta(n^2)$$

- 然后计算每个子数组的和
  - 对于每一个子数组，需要做多少次加法?
    - 取决于子数组的大小: 从0 到 $n - 1$.
    - 至少是 $\Omega(1)$.
- 最后找出最大的和: $\Theta(n^2)$.
- 蛮力法需要 $\Omega(n^2)$ 时间.
- 如何算得更快?

分治算法

- *子问题*: 找出 $A[low .. high]$ 的最大子数组。
  - 参数初始值, $low = 1$, $high = n$.
  - *分解* 将子数组分解成两个大小基本相同的子数组
    - 找到子数组的中间位置 *mid* ，将子数组分成两个更小的子数组 $A[low .. mid]$ 和 $A[mid +1 .. high]$。
  - *求解* 找数组 $A[low .. mid]$ 和 $A[mid +1 .. high]$的最大子数组。
  - *组合* 找出跨越中间位置的最大子数组,
    - 三种情况取和最大的子数组 (跨越中间位置的最大子数组和*求解* 步骤中找到的两个最大子数组)。

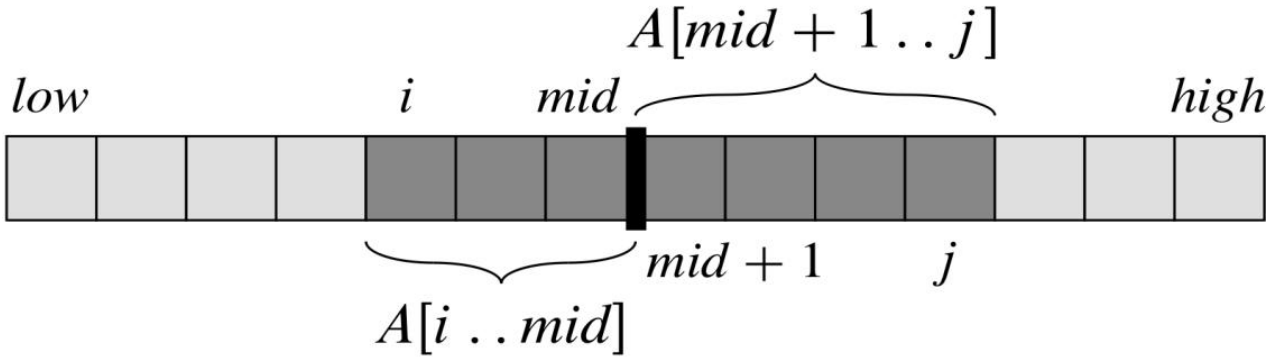
最大子数组问题分治算法

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
  if high == low
       return (low, high, A[low])                    // base case: only one element
  else mid = ⌊(low + high)/2⌋
       (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
       (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
       (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
       if left-sum ≥ right-sum and left-sum ≥ cross-sum
            return (left-low, left-high, left-sum)
       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
            return (right-low, right-high, right-sum)
       else return (cross-low, cross-high, cross-sum)
```

***Initial call*: Find-Maximum-Subarray(*A*, 1, *n*)**

找跨越中间位置的最大子数组

- 子数组必须跨越中间位置。
- 解决思路：
  - 任何一个跨越中间位置 $A[mid]$ 的子数组 $A[i..j]$ 由两个更小的子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成, 其中 $low \le i \le mid < j \le high$.
  - 只要找到最大子数组 $A[i..mid]$ 和 $A[mid+1..j]$，然后把它们合并。
  - 注意：*mid*是固定的，左右分别扫描即可。这个问题可以用 $\Theta(n)$ 时间解决。



算法：找跨越中间位置的最大子数组

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

- *运行时间分析*:
  两个循环总共考虑[***low .. high***]中的每个数组下标一次，每次迭代需要 **Θ(1)** 时间 ➔ 整个过程需要 **Θ(*n*)** 时间.

算法分析

- *简化假设*: 原始问题的规模是2的幂, 所有子问题的规模是整数.
- 用 $T(n)$ 表示最大子数组算法在n个元素数组上的运行时间
- *基本情况*: 当 $high = low$, $n = 1$。算法什么也不做就返回➔ $T(n) = \Theta(1)$。
- *递归情况*: 当 $n > 1$
  - *分解* 需要 $\Theta(1)$ 时间.
  - *求解* 两个子问题, 每个子问题有 $n/2$ 元素， 需要$T(n/2)$ 时间 ➔ 总共需要 $2T(n/2)$ 时间。
  - *合并* 包括调用跨越中间位置最大子数组, 需要 $\Theta(n)$ 时间, 和常数时间的测试 ➔ $\Theta(n) + \Theta(1)$。

算法分析(续)

- 递归情况的递归式

  $T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n)$

- 所有情况的递归式

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

  - 和归并排序的递归式相同.

    ➔ 和归并排序运行时间相同: $T(n) = \Theta(n \lg n)$.

- 最大子数组分治算法运行时间为 $\Theta(n \lg n)$，比蛮力法 $\Omega(n^2)$ 快。

Strassen矩阵乘法

- 蛮力法：$O(n^3)$
- 分治法:
  - 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：

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

注意：只关心乘法执行次数

Strassen矩阵乘法

- 为了降低时间复杂度，必须减少乘法的次数。

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

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

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

Strassen矩阵乘法

- 时间的递推关系式
  - 当n>1时，$M(n)=7M(n/2)$，$M(1)=1$
  - 因为$n=2^k$，$M(n)=7M(n/2)=7^2M(n/2^2)\cdots\cdots$
    $=7^kM(1)=7^k$
  - $M(n)=7^{\log_2 n}=n^{\log_2 7}\approx n^{2.807}$

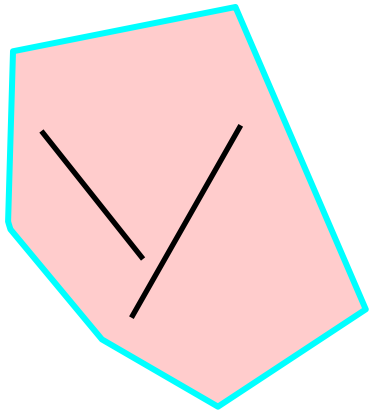
➢Hopcroft和Kerr已经证明(1971)，计算2个 2 × 2 矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究 3 × 3 或 5 × 5 矩阵的更好算法。

➢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 $O(n^{2.376})$

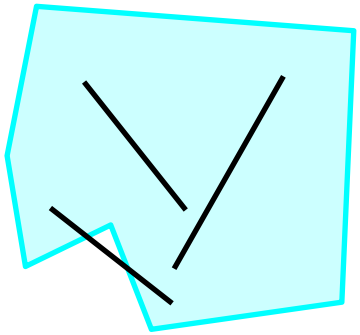
➢是否能找到理论下界$O(n^2)$的算法？？？目前为止还没有结果。

凸包问题

- 定义 对于平面上的一个点集合（有限或无限），如果以集合中任意两点P和Q为端点的线段都属于这个集合，则这个集合是*凸*的。

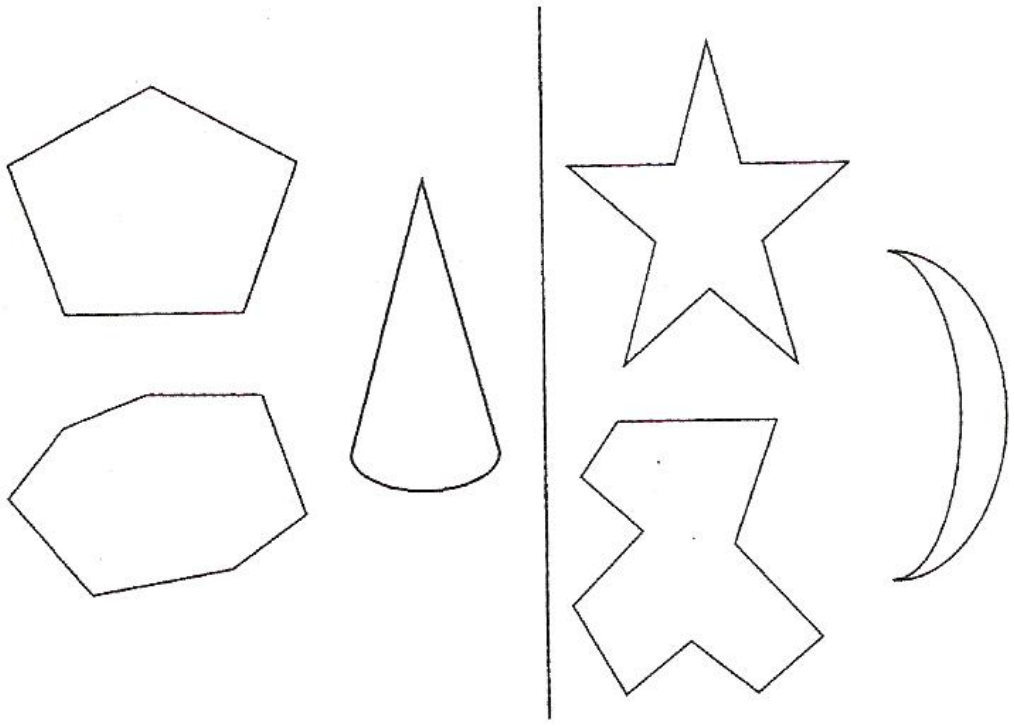


convex



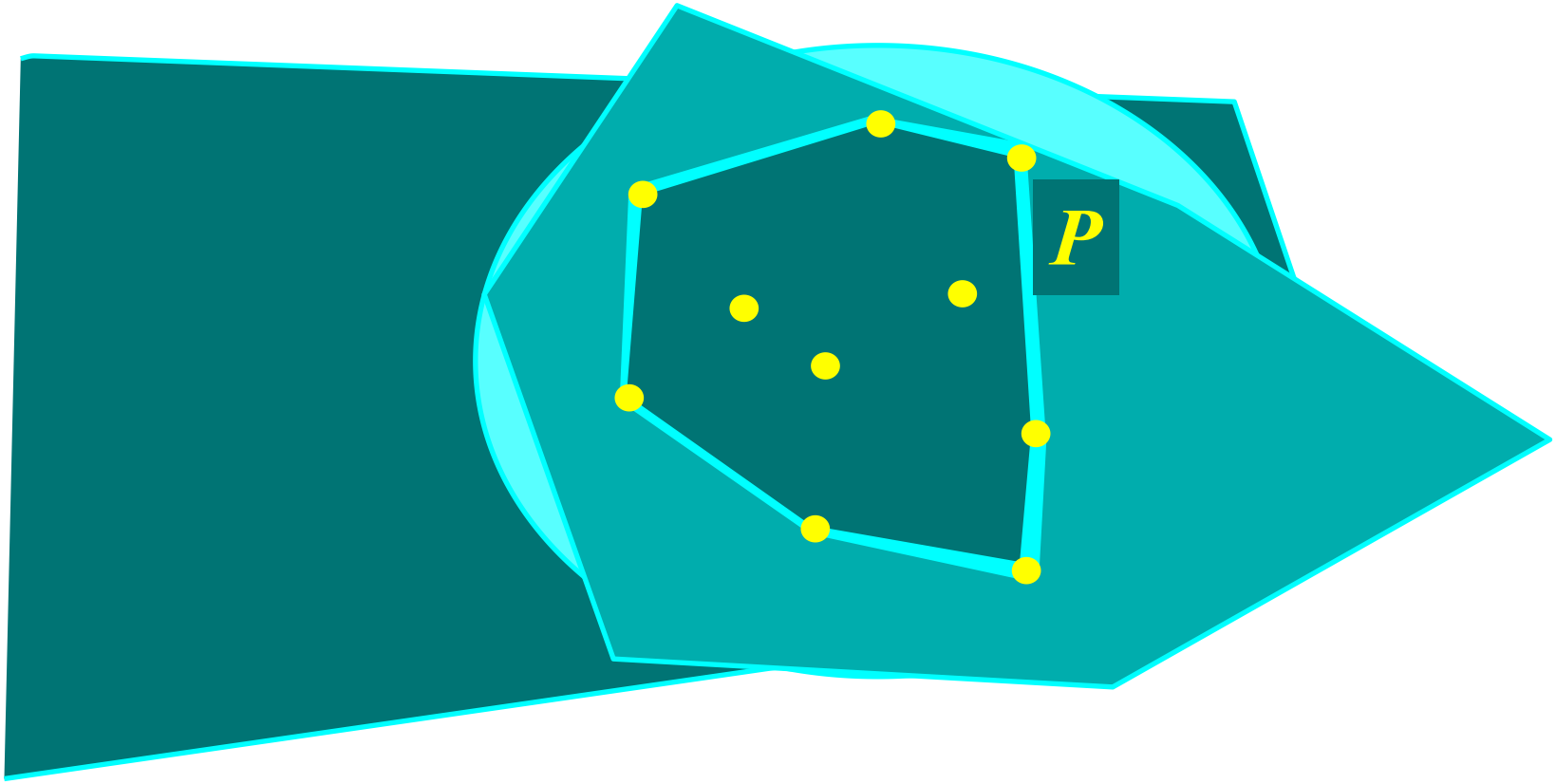
non-convex

凸包问题



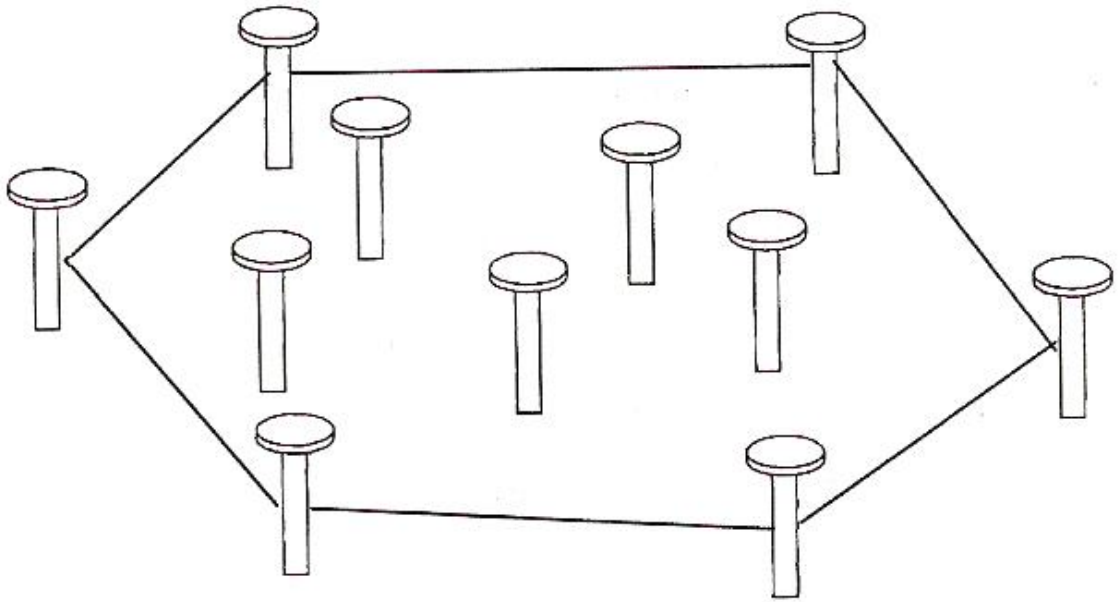
凸包问题

- 定义 一个点集合S的凸包是包含S的最小凸集合。



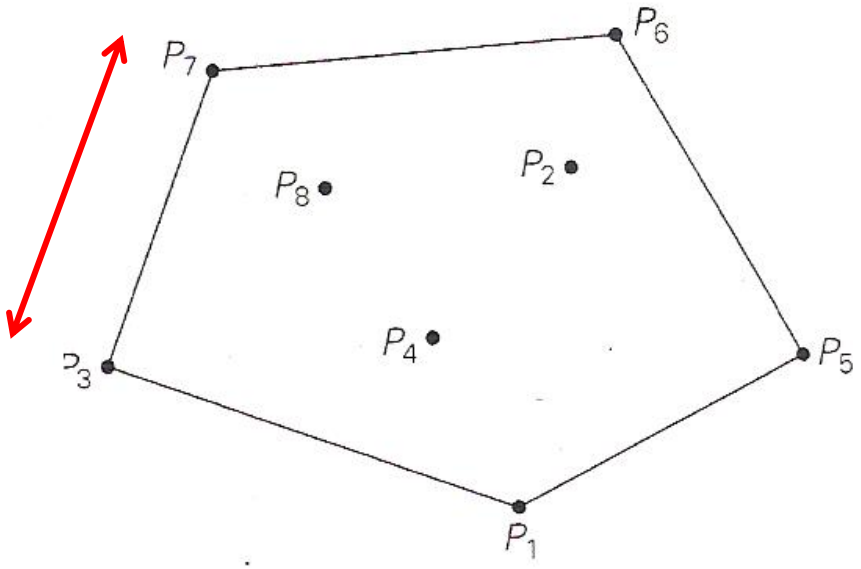
凸包问题

- 通俗理解：用皮筋绑定点集边缘，皮筋内的区域就是凸包。



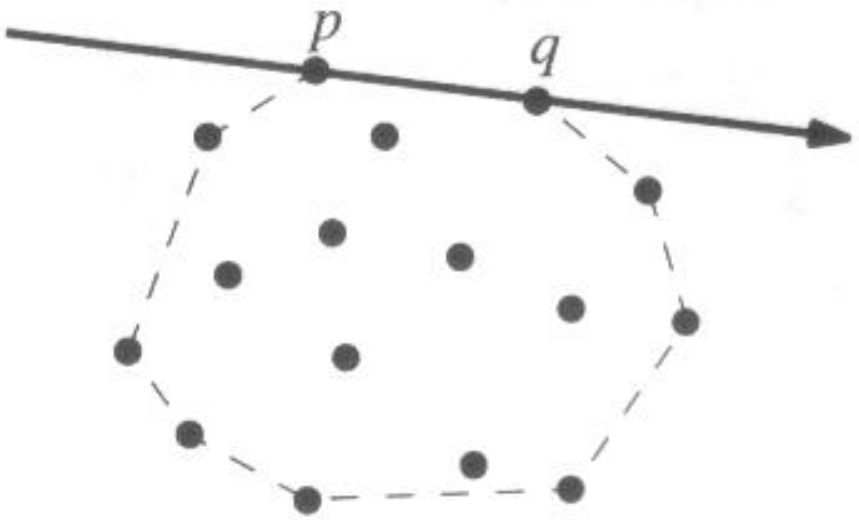
凸包问题

- **定理：** 任意包含n>2个点（不共线）的集合S的凸包是以S中的某些点为顶点的凸多边形。
- 凸包问题是为一个n个点的集合构造凸包的问题。
- 极点：对于任何以集合中的点为端点的线段来说，它们不是这种线段的中点。

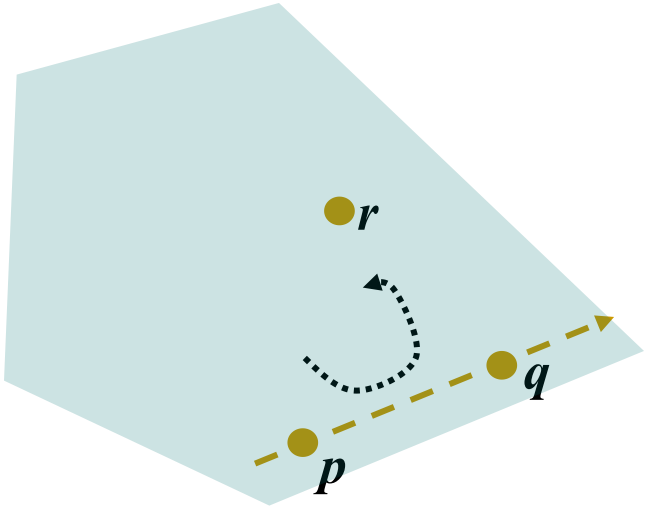


凸包问题——蛮力法

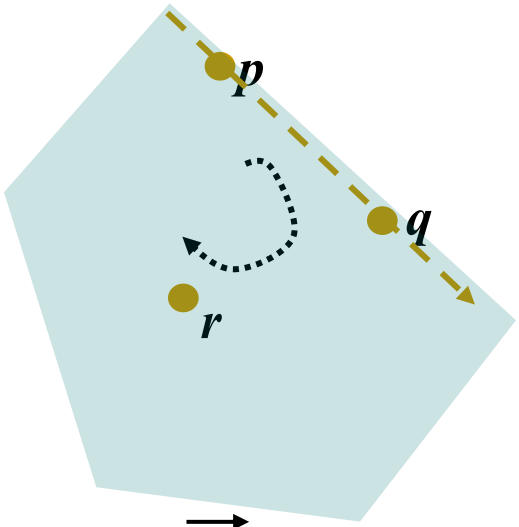
- 对于一个n个点集合中的两个点P_i和P_j，当且仅当该集合中的其他点都位于穿过这两点的直线的同一边时它们的连线是该集合凸包边界的一部分。对每一对点都做一遍检验之后，满足条件的线段构成了该凸包的边界。
- 时间效率：$\mathbf{O(n^3)}$



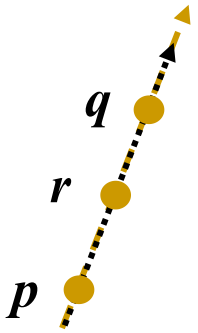
在直线一侧如何检测（方向检测）



•*r* 在$\overrightarrow{pq}$左侧



•r 在$\overrightarrow{pq}$右侧



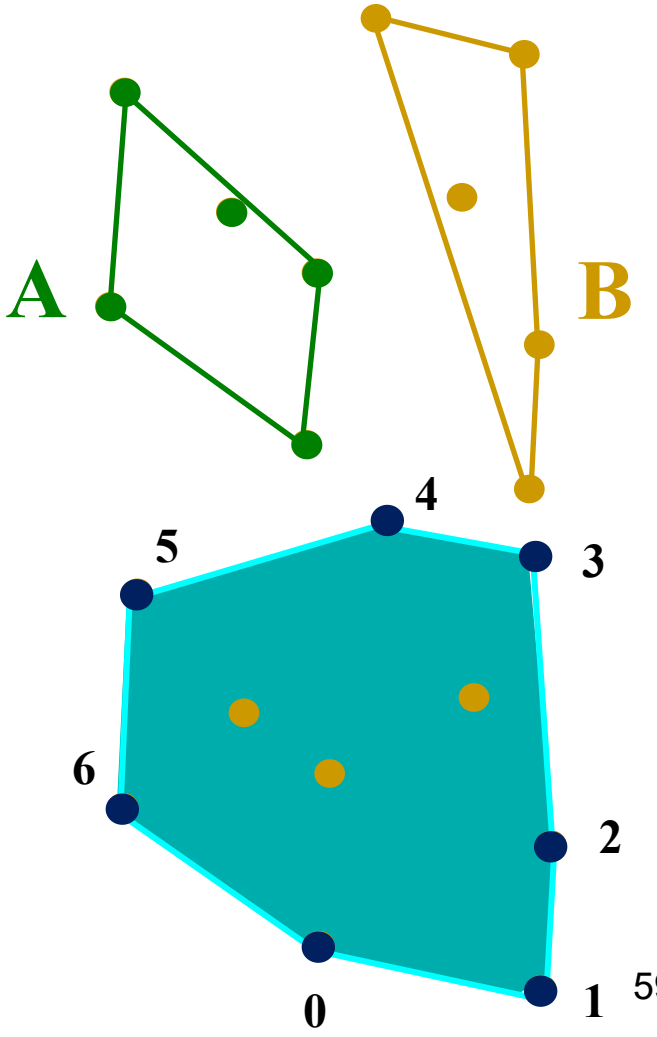
•r 在 $\overrightarrow{pq}$上

行列式 $\begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}$,其中 $p = (p_x, p_y)$

- 行列式的正负
- 常数时间完成

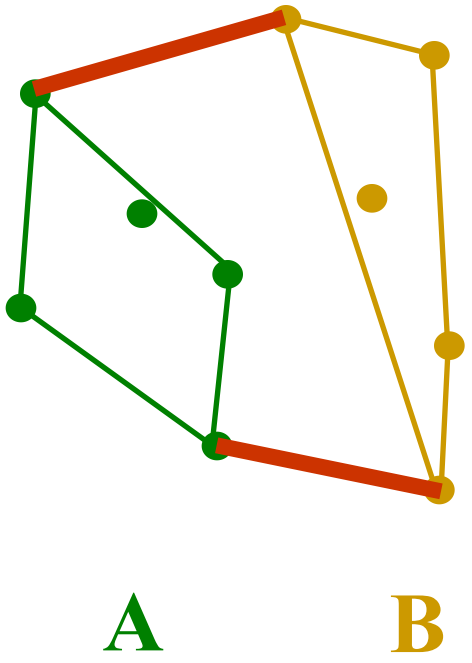
凸包问题(Convex Hulls Problem)

- **分解**：将集合S中的点按X轴坐标升序排列，用竖线将点集分成两个子集A和B。
- **求解**：
  - 递归求解A的凸包；
  - 递归求解B的凸包；
- **合并**：合并两个凸包。
- 注：凸包中的点用逆时针编号



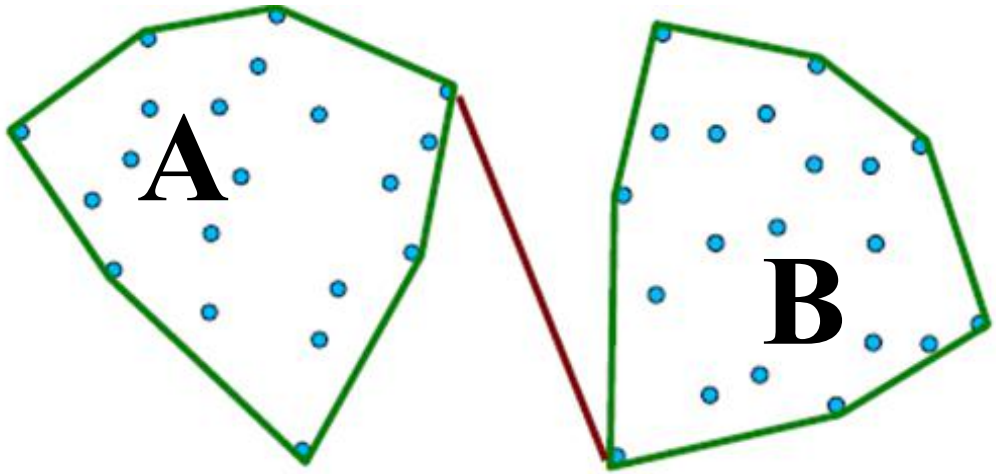
合并

- 找到最上面的切线和最下面的切线
- 确定最终的凸包范围

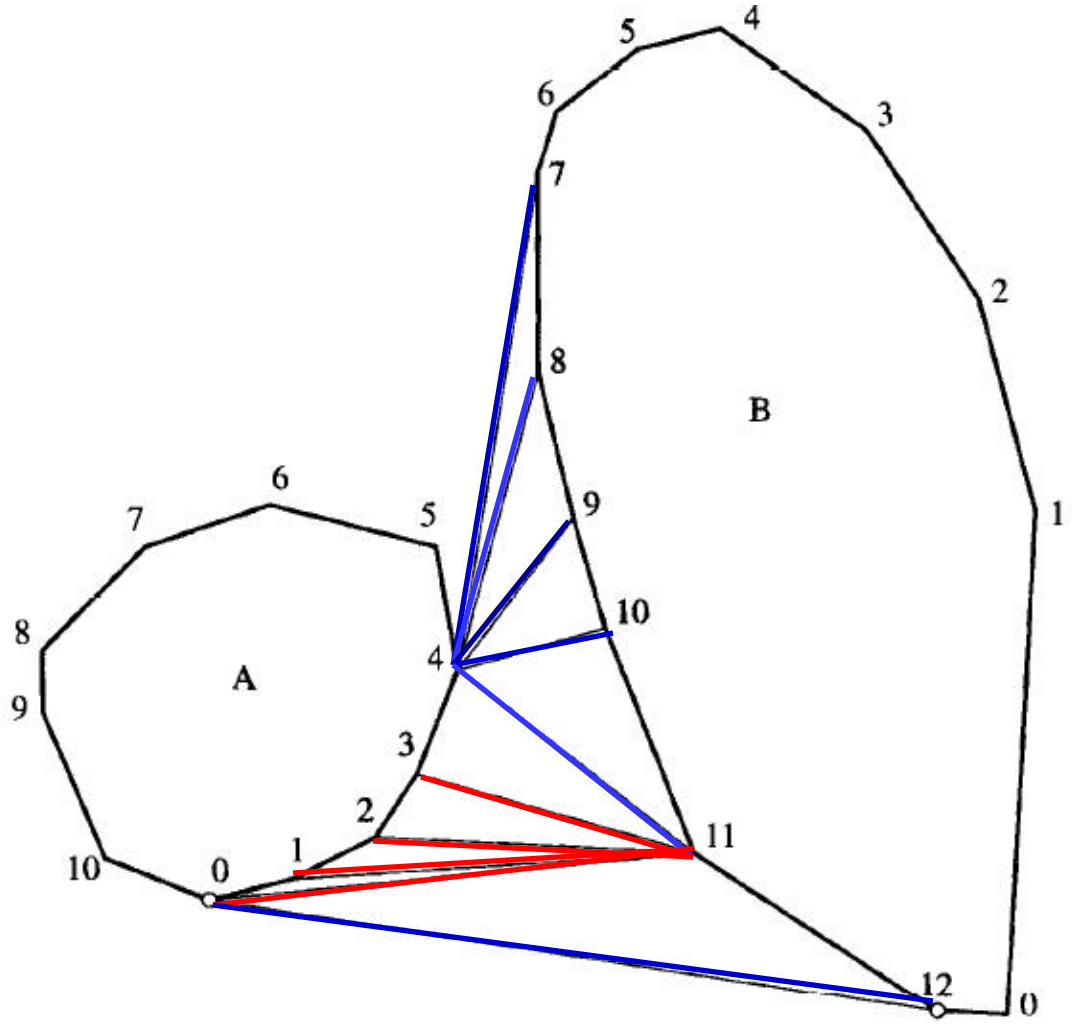


寻找下切线

- 从A集合的最右边的点和B集合最左边的点开始
- 如果这条线不是A和B的下切线，则：
  - 如果不是A的下切线，则逆时针旋转；
  - 如果不是B的下切线，则顺时针旋转；



下切线举例



- A中最右边的点是4，B中最左边的点是7
- T=(4, 7) 是A的下切线，a循环不执行，但是不是B的下切线，b循环增至11
- T=(4, 11) 不是A的下切线，a循环至
- T=(0, 11) 不是B的下切线，b 增至12.
- T=(0, 12) 是A 和 B的下切线，返回T

找最下面的切线

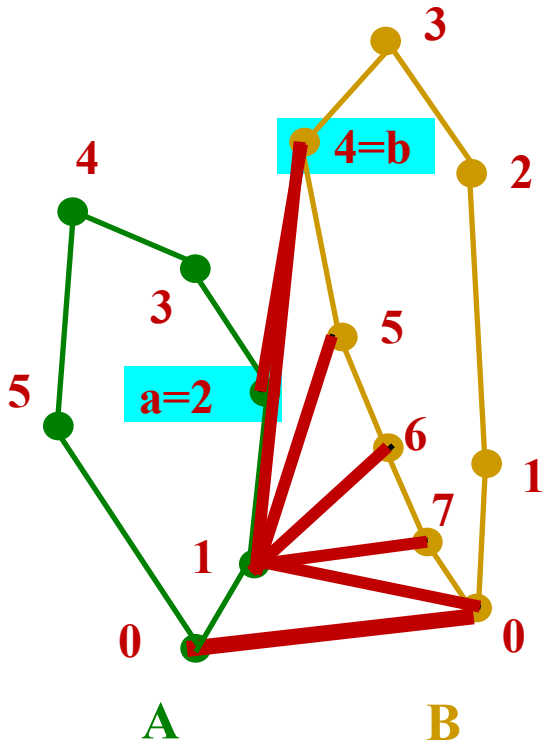
a = A中最右边的点

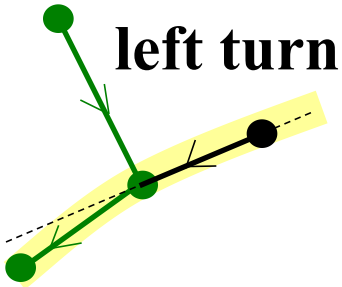
b = B中最左边的点　　T=ab

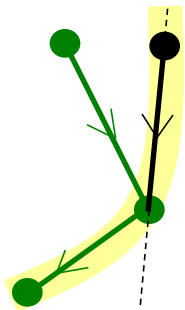
```
while T not lower tangent to both
      convex hulls of A and B do{

    while T not lower tangent to
     convex hull of A do{
        a=a-1
    }
    while T not lower tangent to
     convex hull of B do{
        b=b+1
     }
}
```

相邻顶点做方向检测（常数时间完成）




left turn


right turn

凸包问题运行时间

- 递归关系式:

  $T(n) = 2\,T(n/2) + cn$

- 解：$T(n) = \Theta(n \log n)$

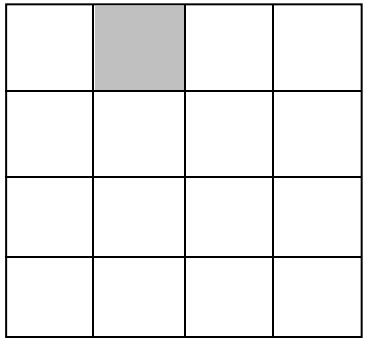
凸包问题(Convex Hulls Problem)

Algorithm	Speed	Discovered By
Brute Force	$O(n^3)$	[Anon, the dark ages]
Gift Wrapping	O(nh)	[Chand & Kapur, 1970]
Graham Scan	O(n log n)	[Graham, 1972]
Jarvis March	O(nh)	[Jarvis, 1973]
QuickHull	O(nh)	[Eddy, 1977], [Bykat, 1978]
Divide-and-Conquer	O(n log n)	[Preparata & Hong, 1977]
Monotone Chain	O(n log n)	[Andrew, 1979]
Incremental	O(n log n)	[Kallay, 1984]
Marriage-before-Conquest	O(n log h)	[Kirkpatrick & Seidel, 1986]

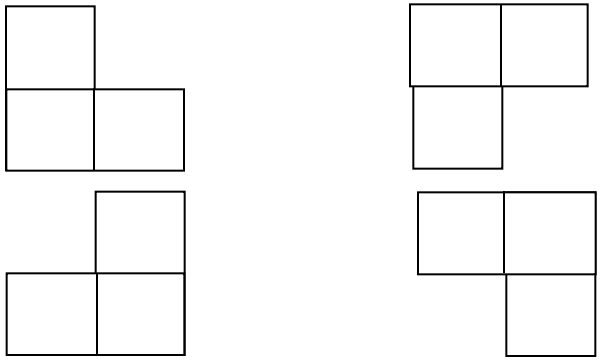
棋盘覆盖问题

在一个$2^k \times 2^k$（$k \geq 0$）个方格组成的棋盘中，恰有一个方格与其他方格不同，称该方格为特殊方格。

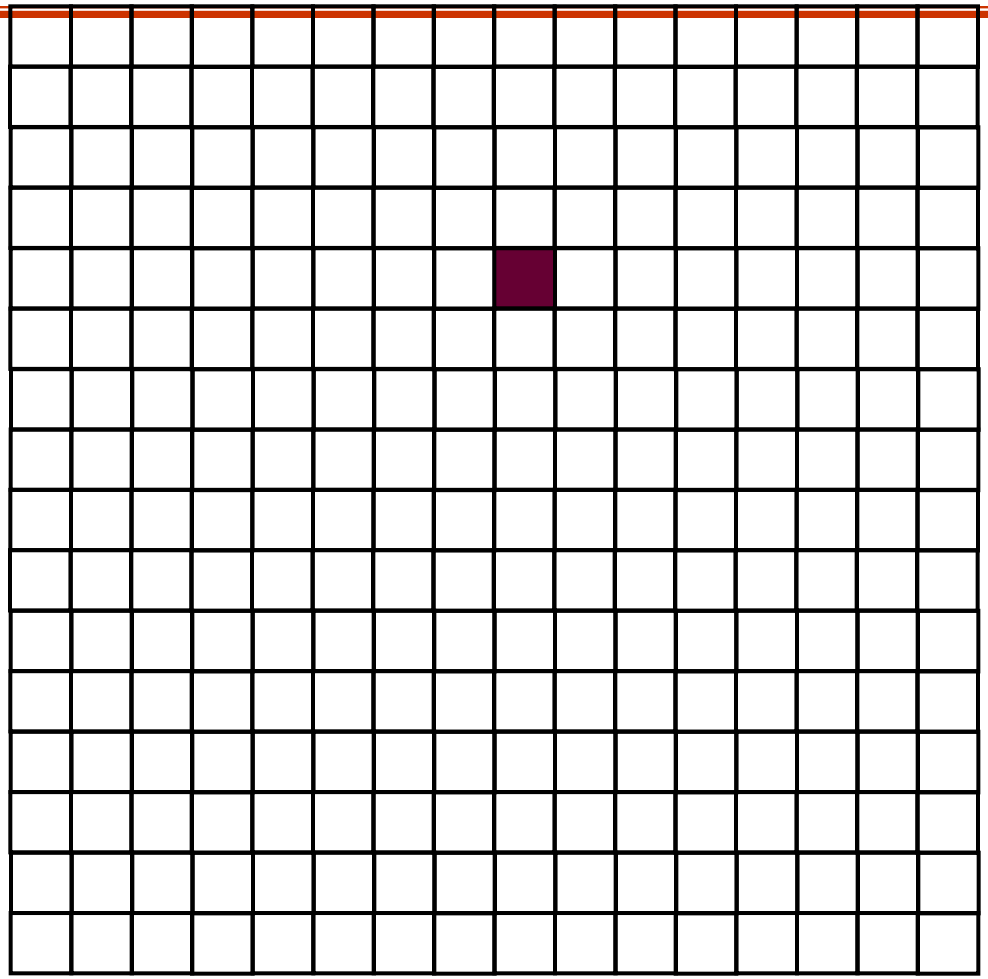
棋盘覆盖问题要求用如图（b）所示的L型骨牌覆盖给定棋盘上除特殊方格以外的所有方格，且骨牌之间不得有重叠。

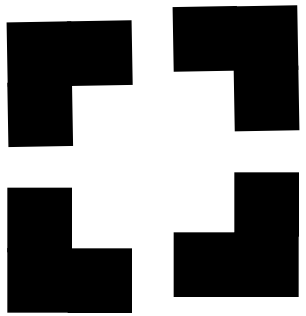


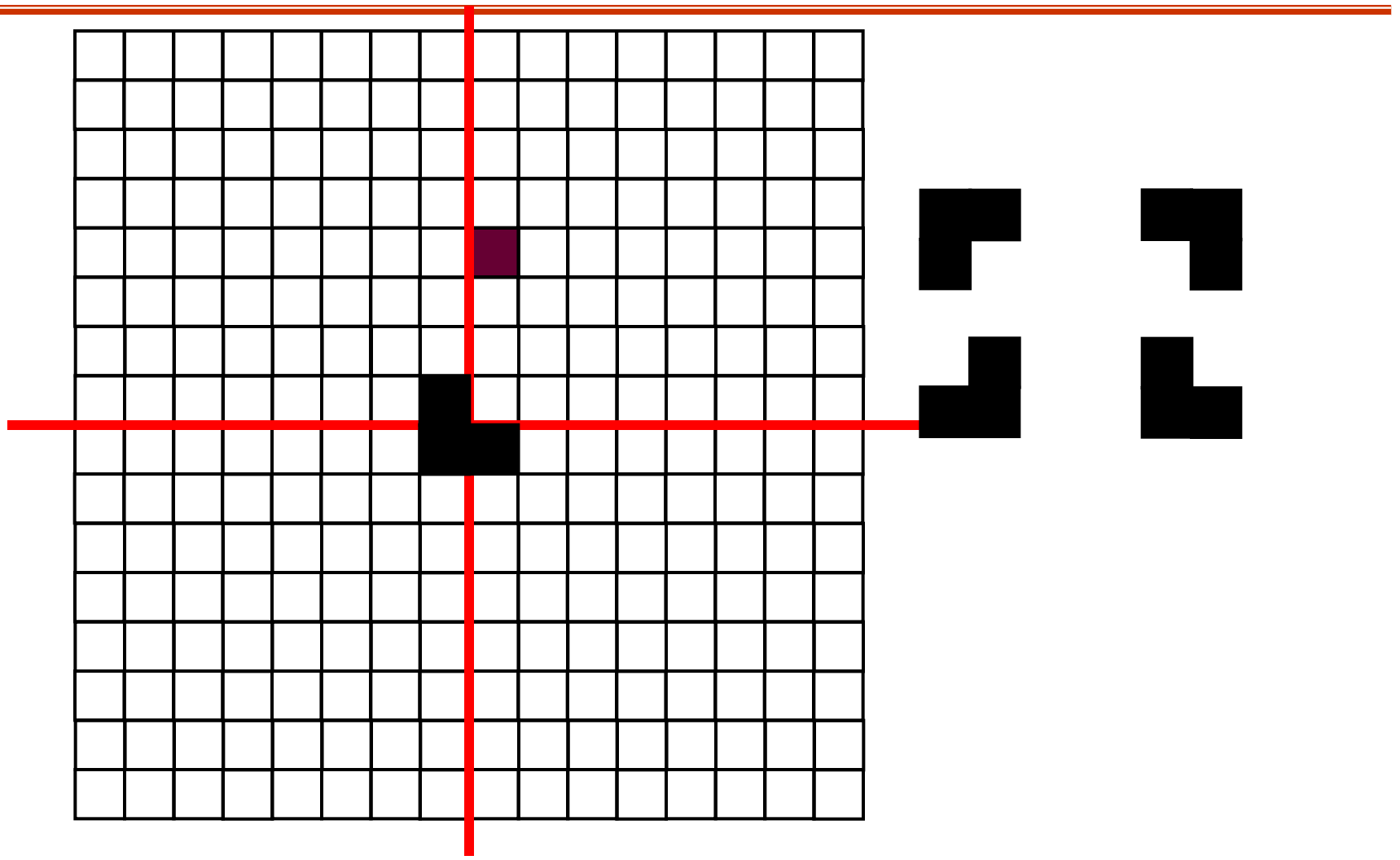
(a) k=2时的一种棋盘

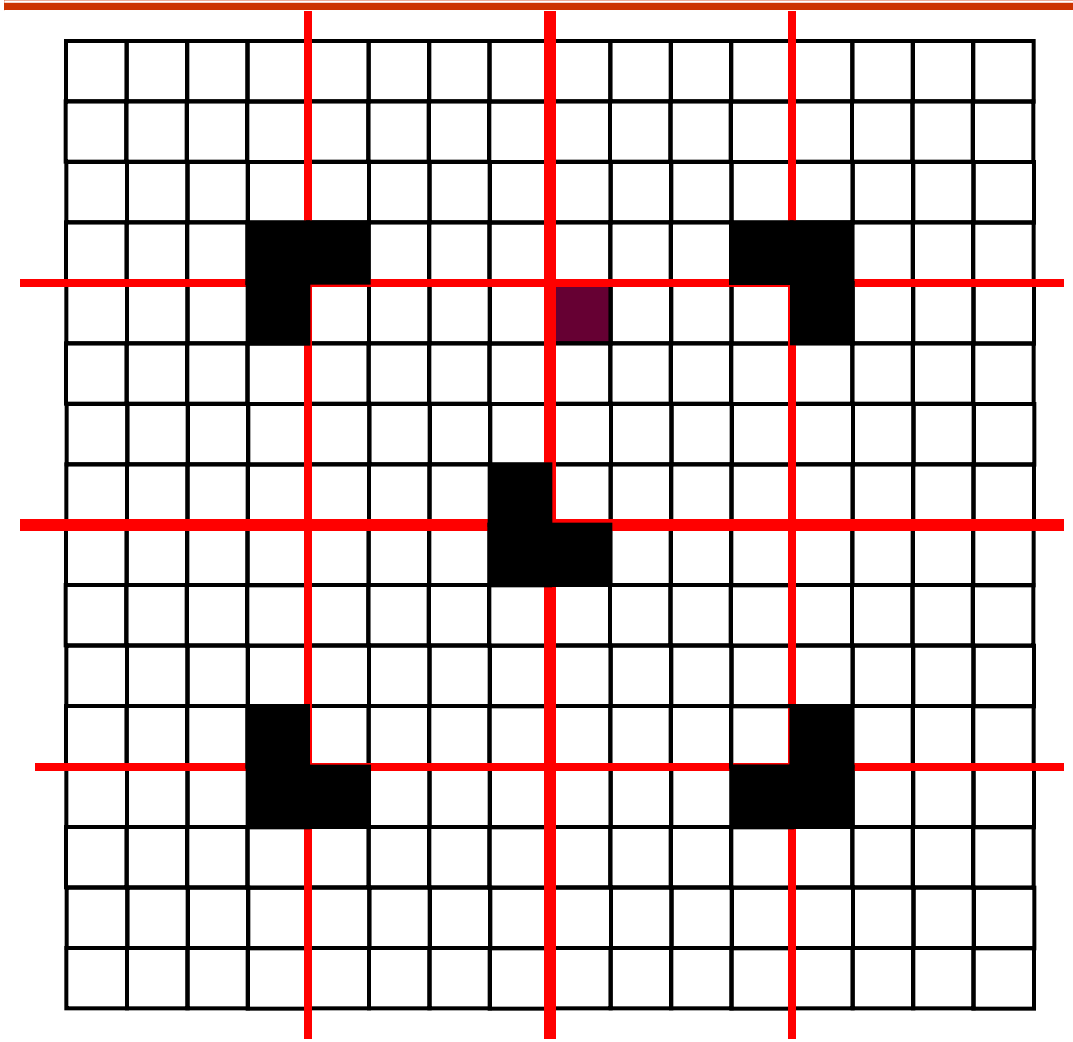


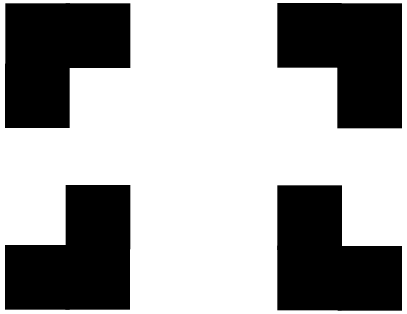
(b) 4种不同形状的L型骨牌

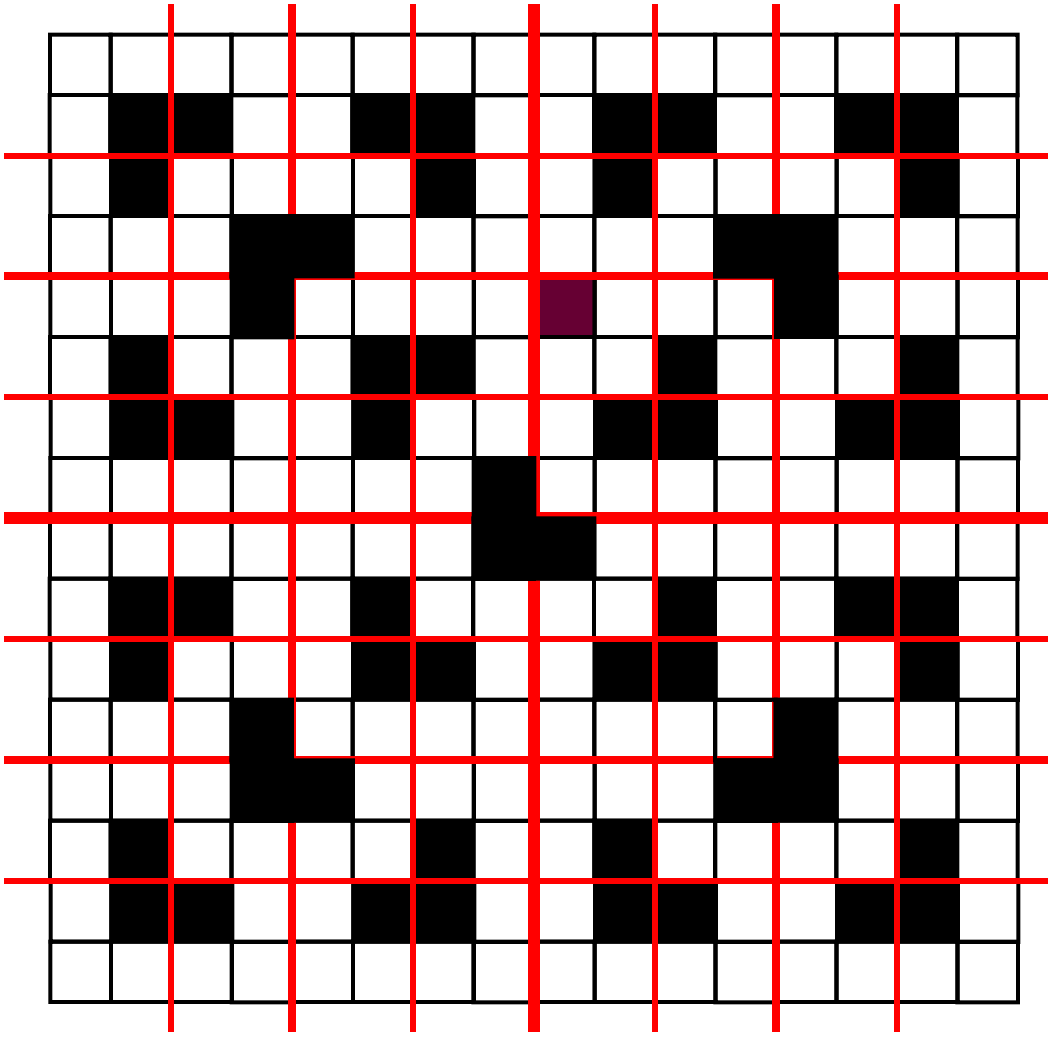


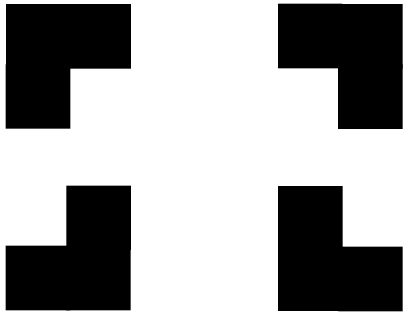


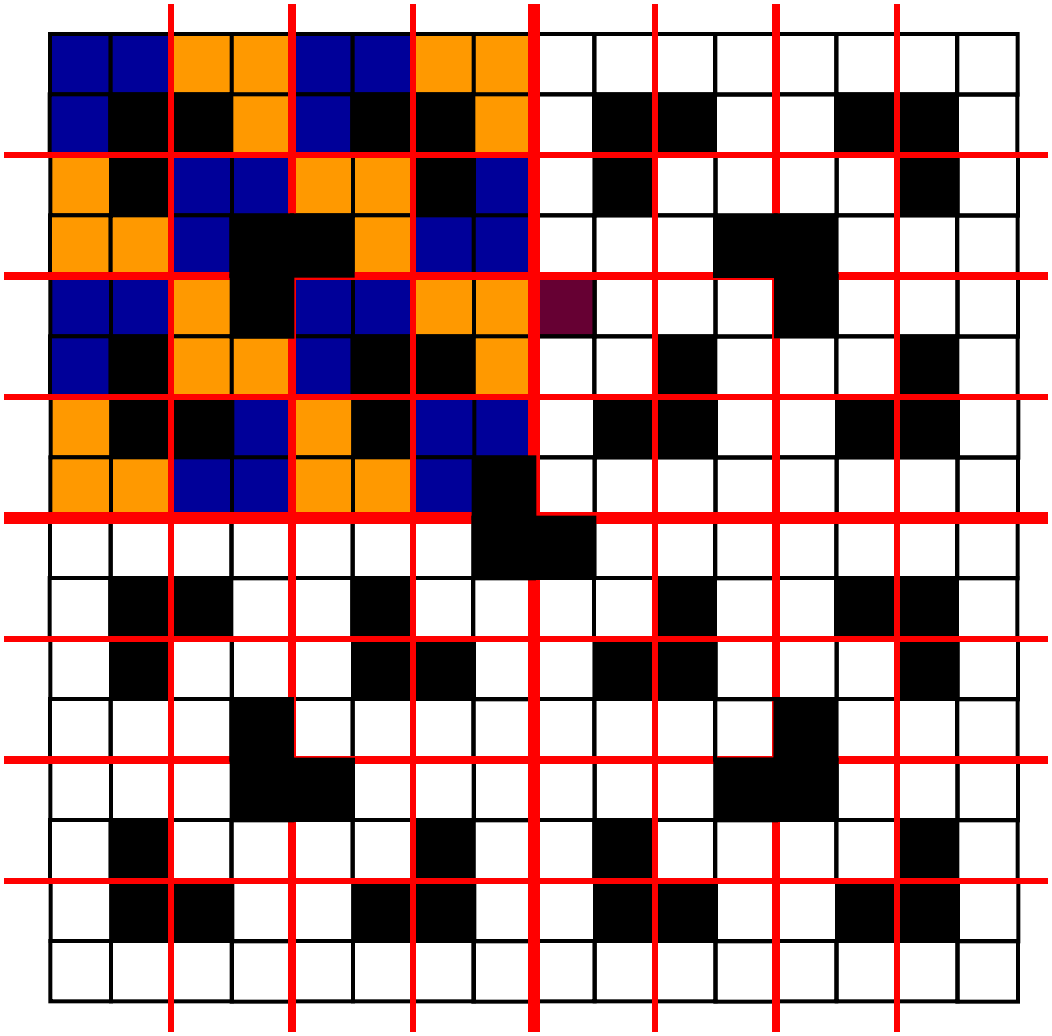


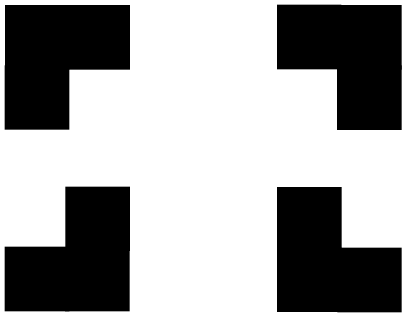


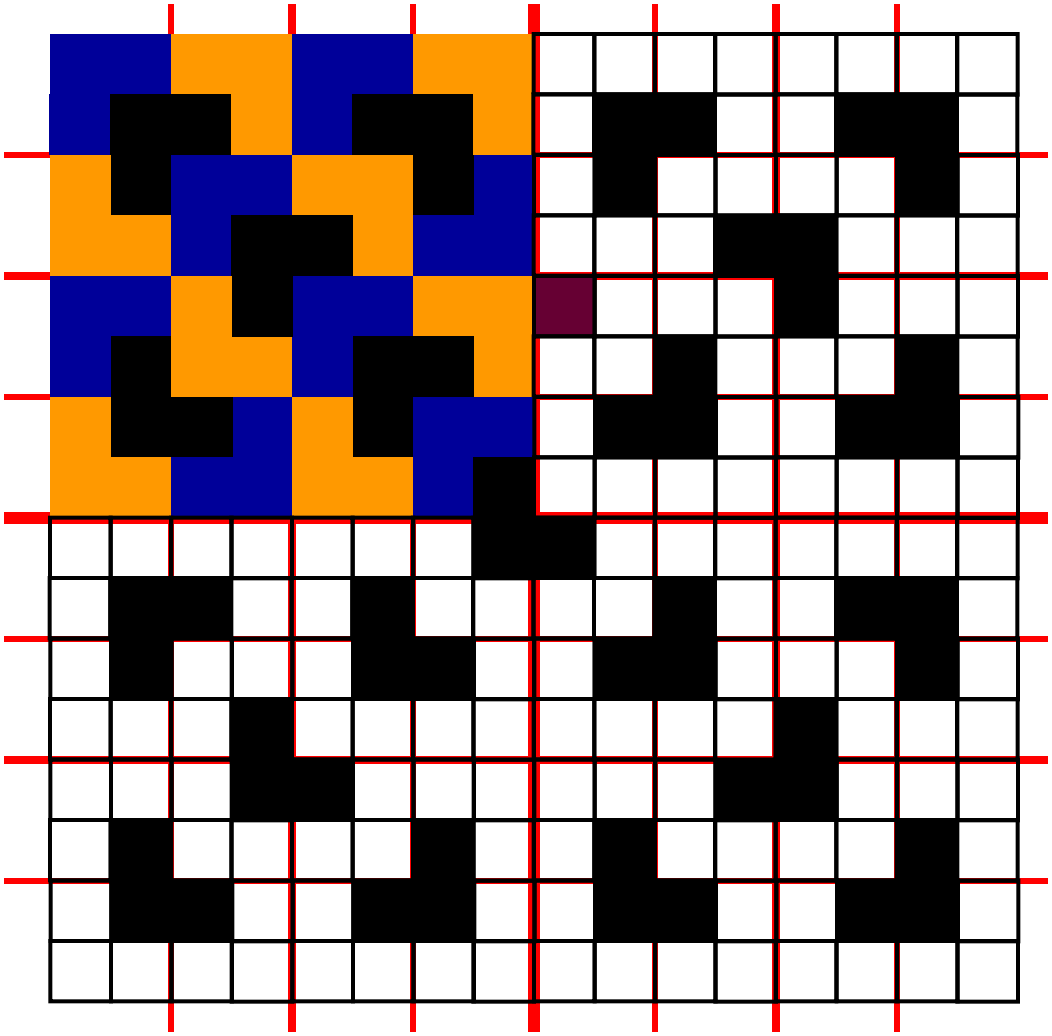


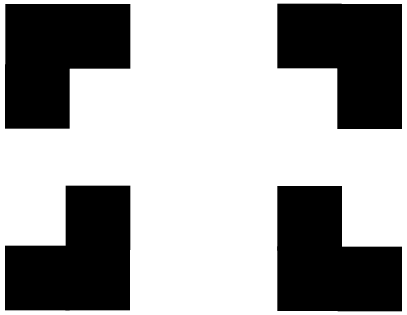








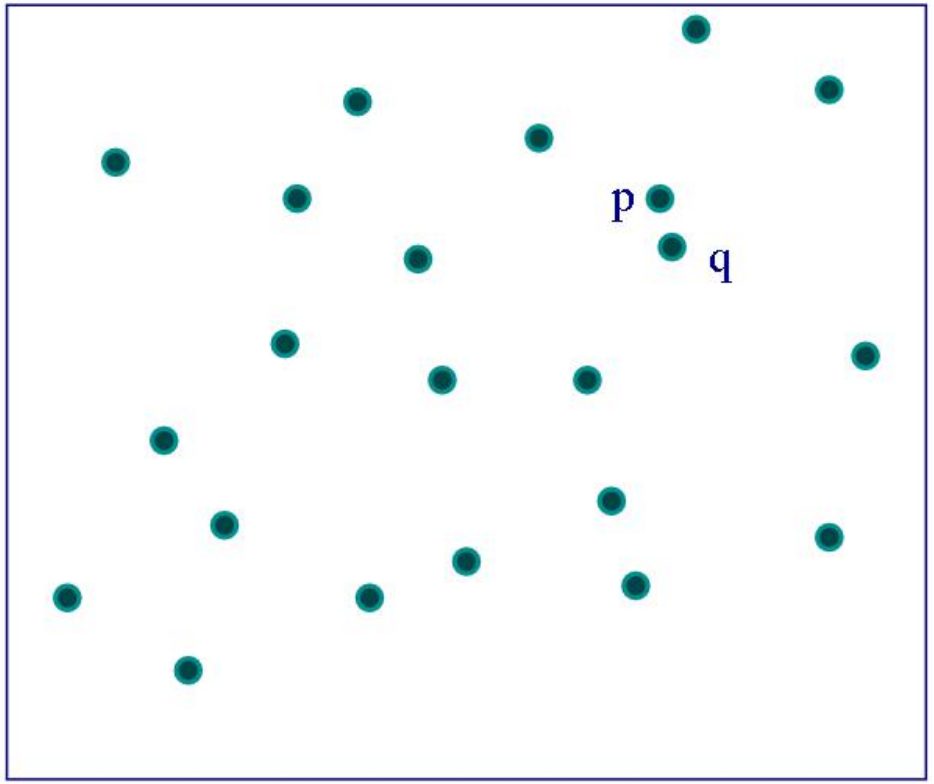




二维最近对问题

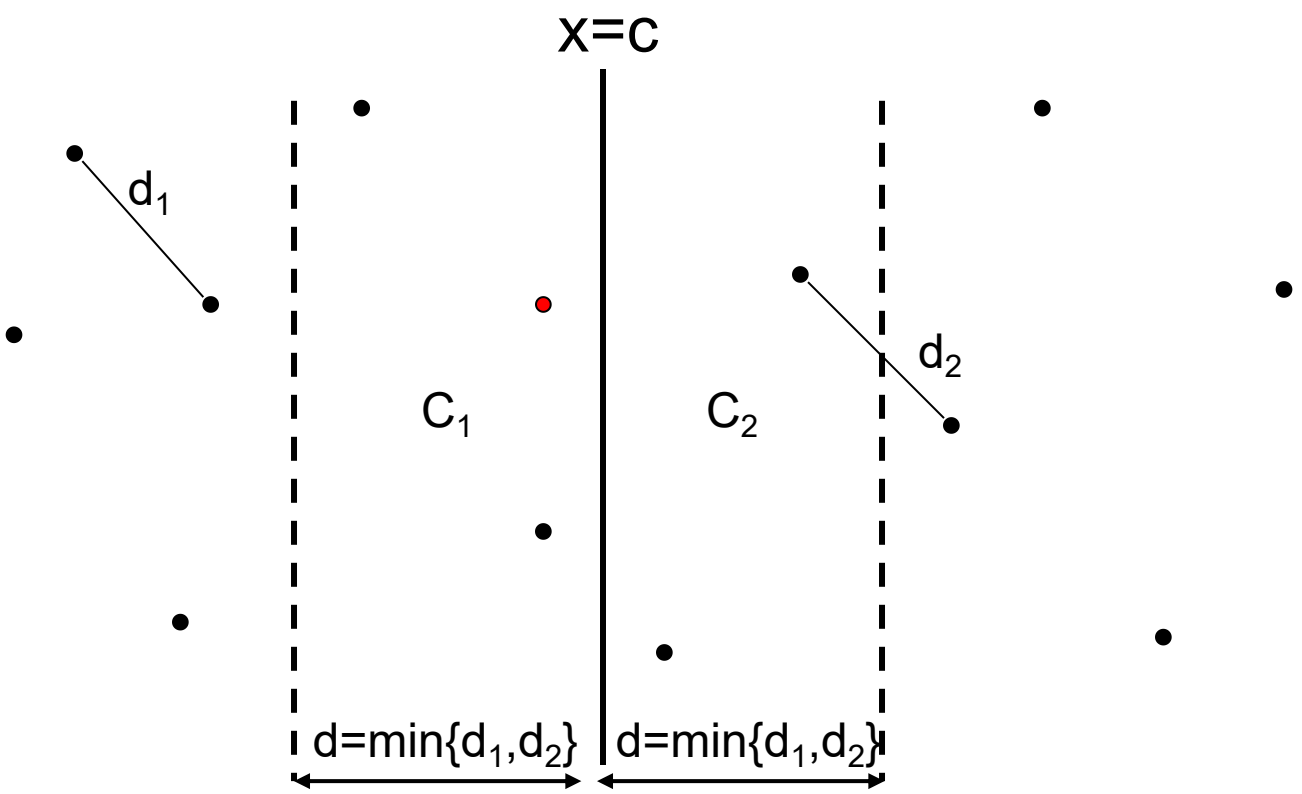
- $P_1(x_1,y_1),\ldots,P_n=(x_n,y_n)$ 是平面上n个点构成的集合S，假设$n=2^k$，最近点对问题要求找出距离最近的两个点

- 最近点对问题是许多算法的基本步骤



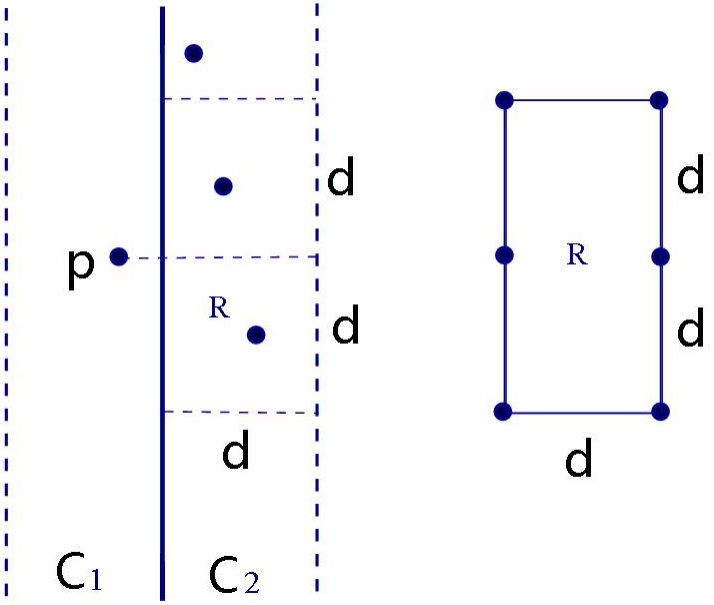
二维最近点对问题

- 将点集S分为S_1和S_2，分隔线是S在x轴的中点（如何确定x=c?）
- 递归求解S_1和S_2的最近点对，令$d=\min\{d_1,d_2\}$，确定C_1和C_2
- 将C_1和C_2的最近点对合并



二维最近对问题

- 在合并两个子集C_1和C_2时，对于C_1中的每个点P(x,y)，都须要检查C_2中的点和P之间的距离是否小于d。
- 假设p在C_1中，在C_2中与p距离小于d的点不会超过六个
- 最多进行6*n/2次比较



二维最近对问题

计算时间：

合并最小问题所花的时间为$M(n)=O(n)$

该算法的最差递归时间为：

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