分治法

深圳大学计算机与软件学院
卢亚辉

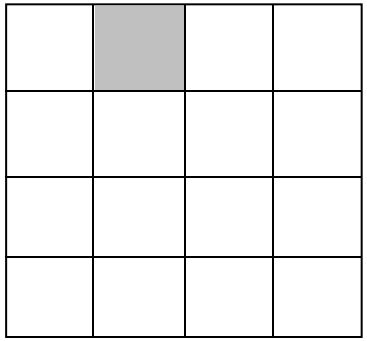
主要内容

- 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

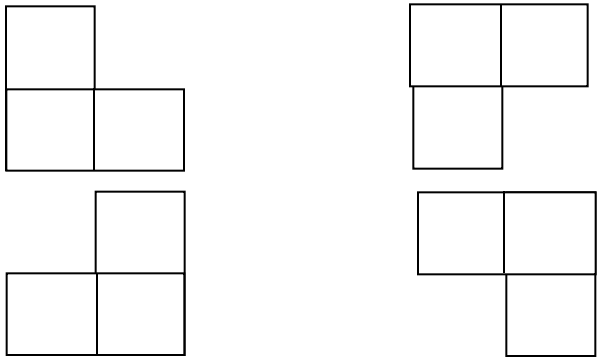
棋盘覆盖问题

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。

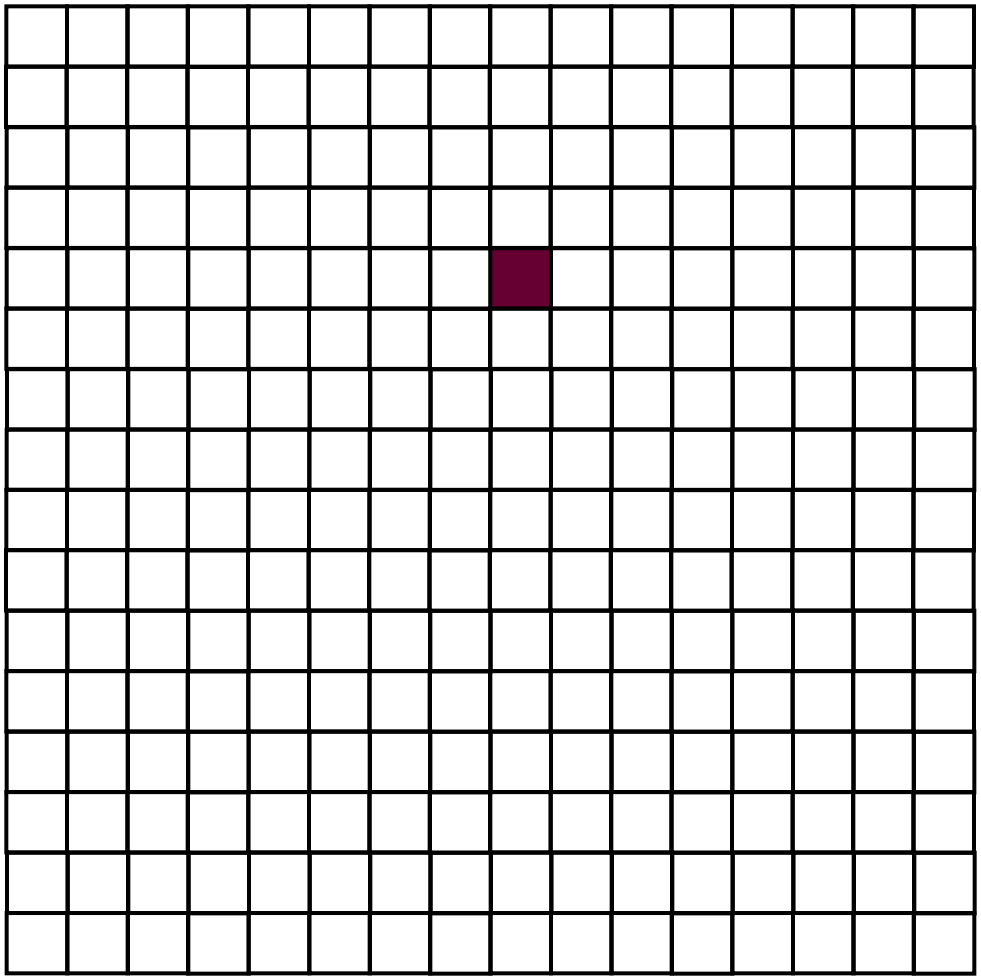
棋盘覆盖问题要求用如图（b）所示的L型骨牌覆盖给定棋盘上除特殊方格以外的所有方格，且骨牌之间不得有重叠。

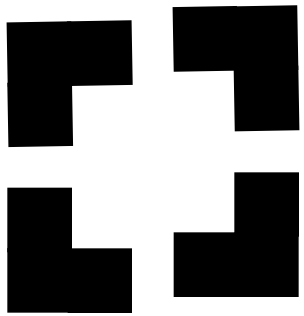


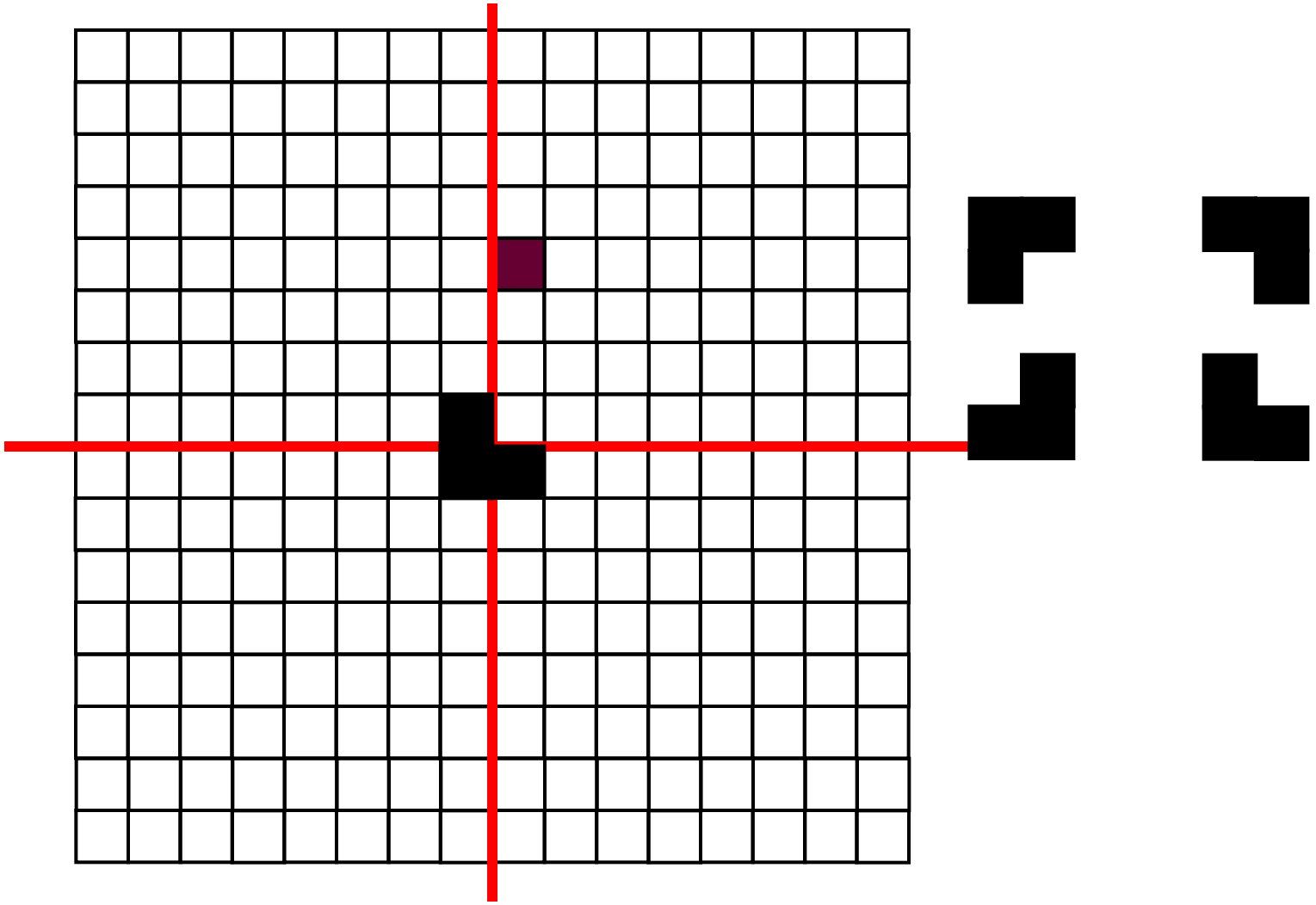
(a) k=2时的一种棋盘

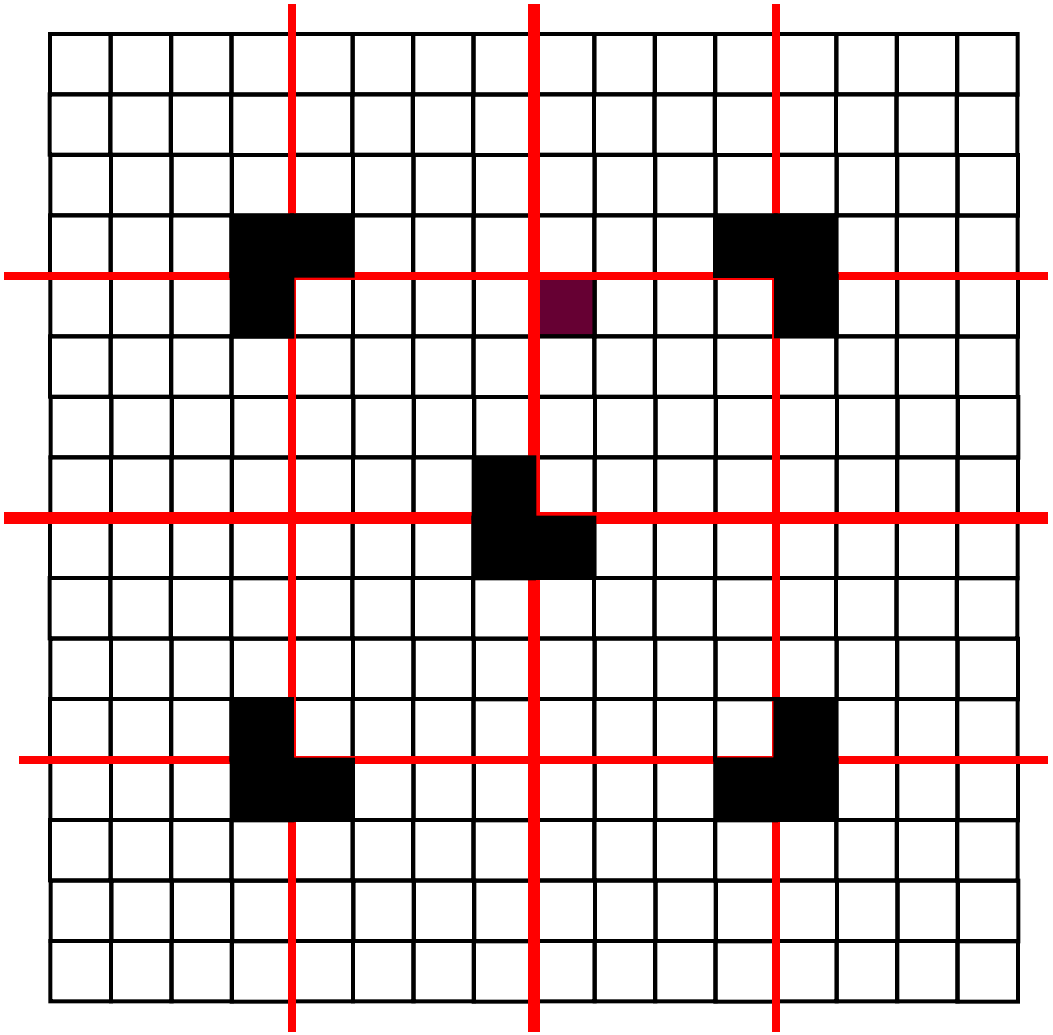


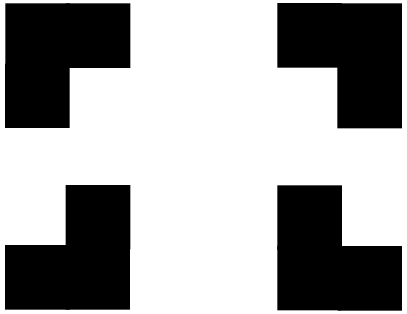
(b) 4种不同形状的L型骨牌

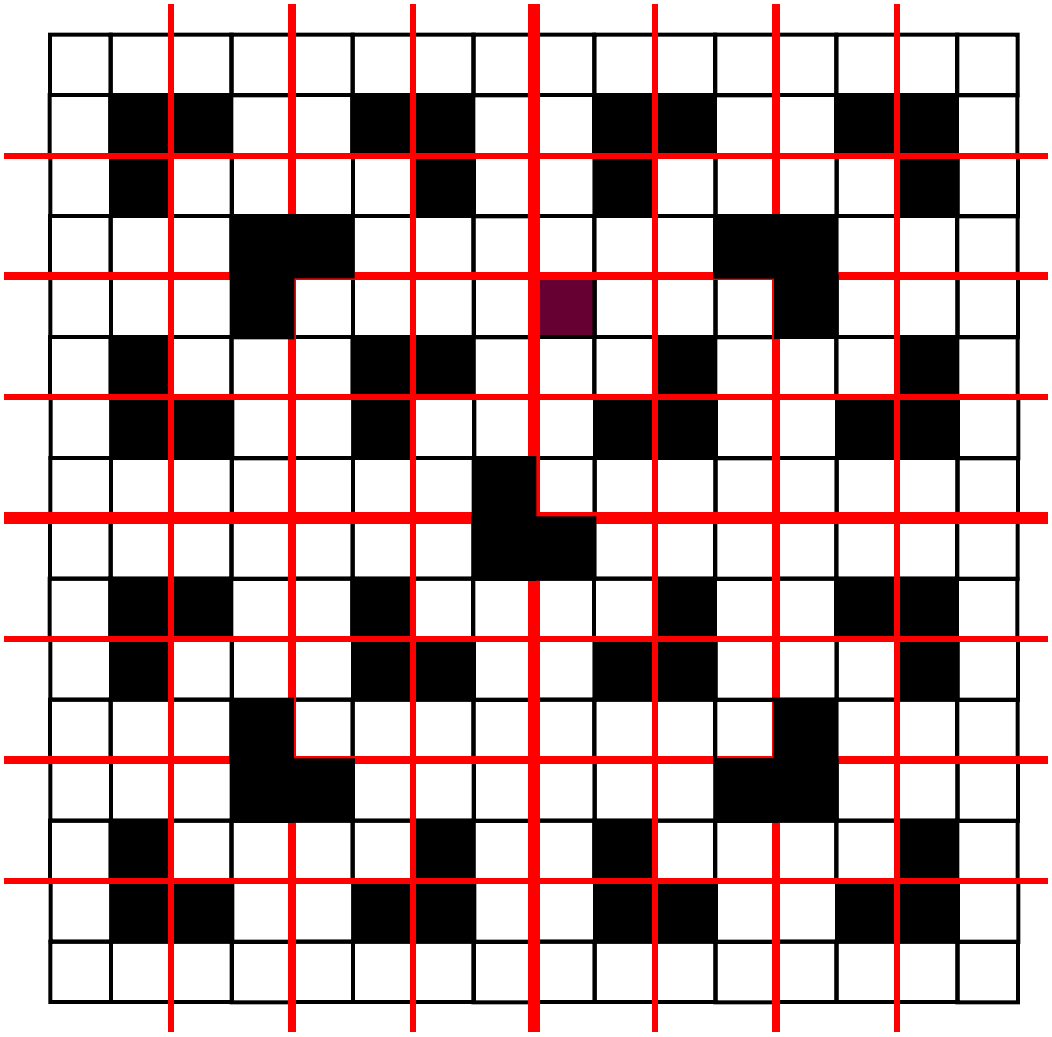


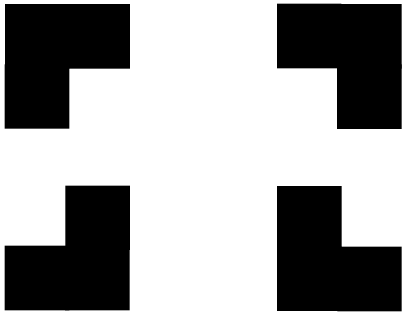


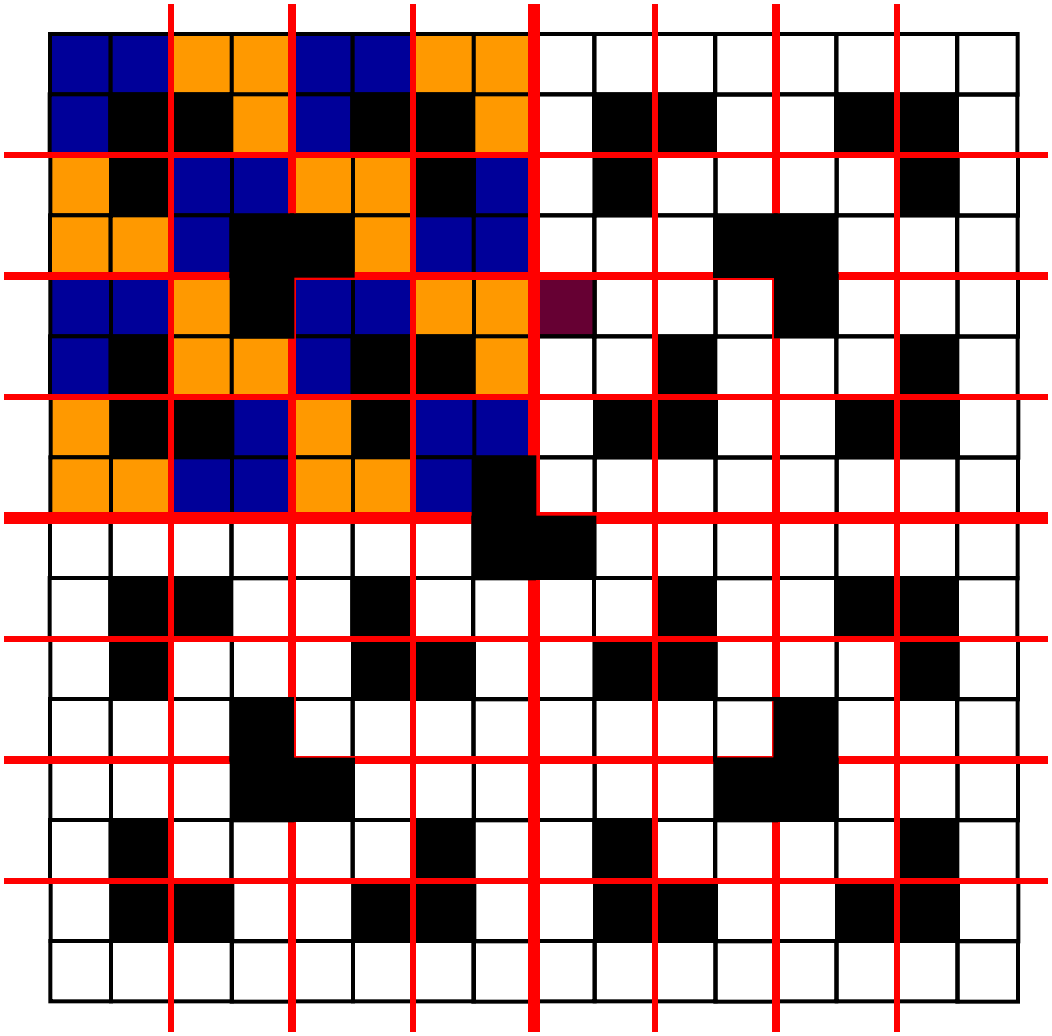


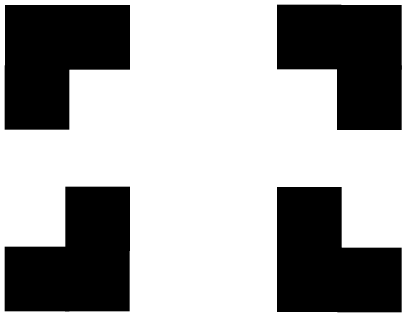


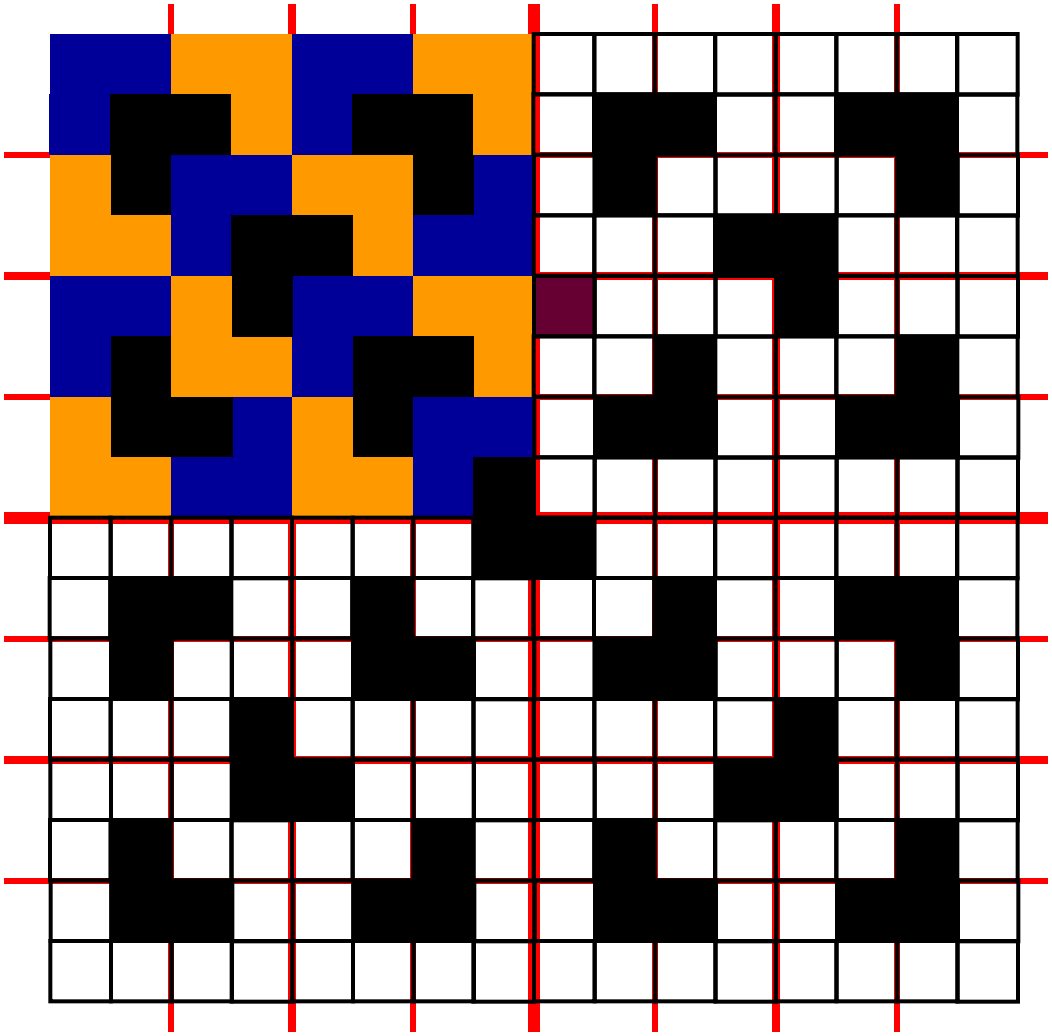


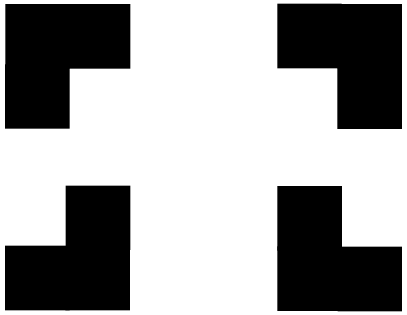








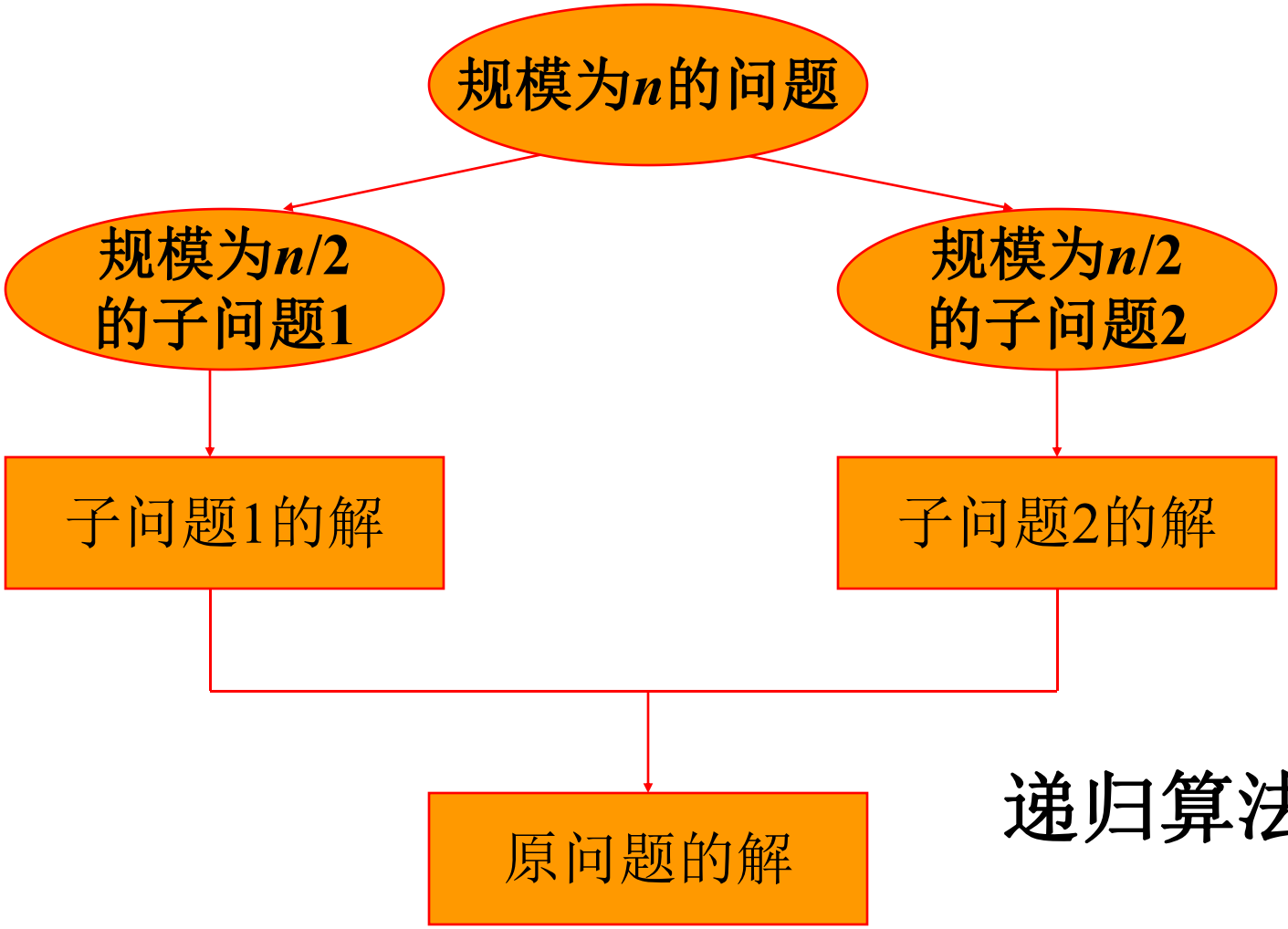




分治策略

- 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



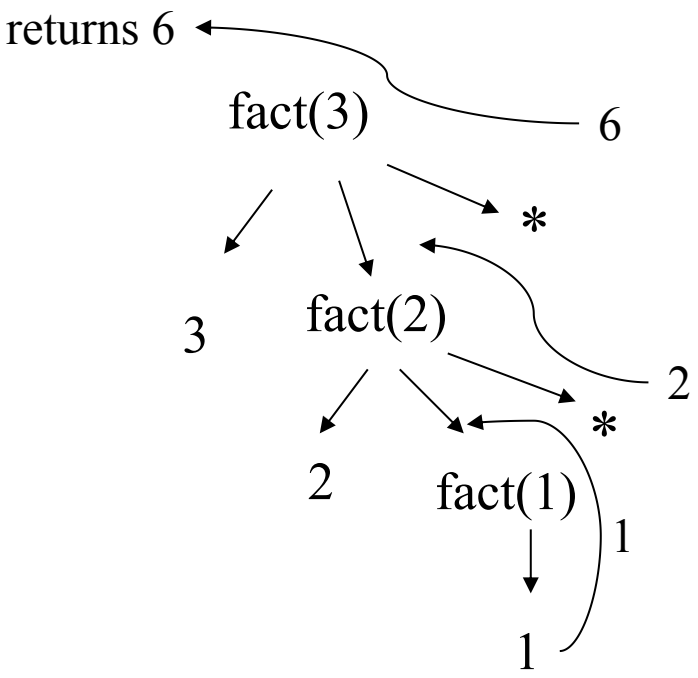
递归算法！

递归算法

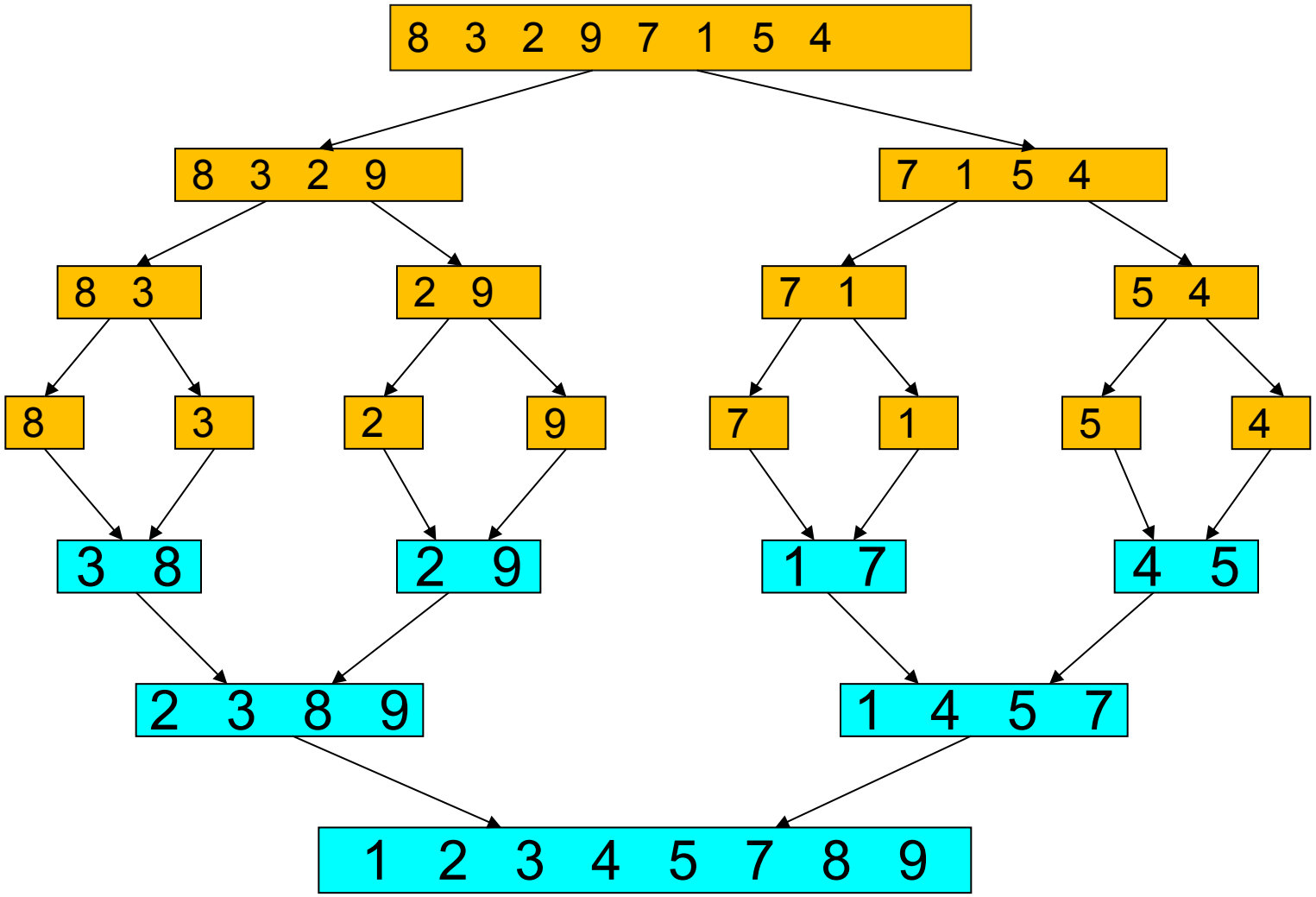
- 一个递归算法通常包含递归的调用该算法本身，传入较小的参数。
- 递归算法的中止条件：
  - 处理基本情况，这些情况不可以有任何递归调用。

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 $A[p \,..\, r]$

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

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 $A[p \,..\, r]$

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      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(2)

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

$if \quad n=2^k$

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

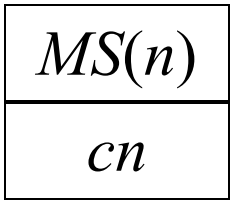
$$\therefore T(n)\in\Theta\left(n\log n\right)$$

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)。

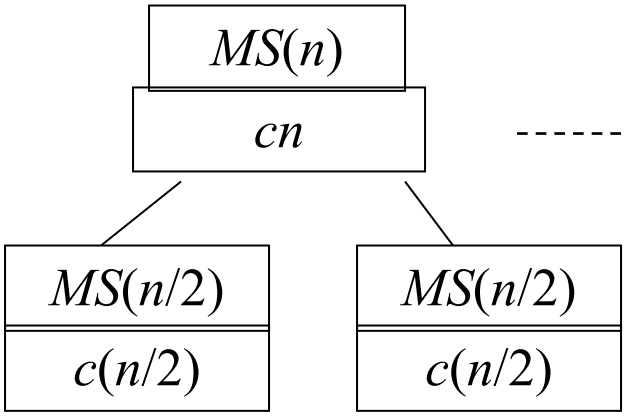
  例如：T(n) = 2T(n/2) + cn, 非递归部分为 cn， cn就是节点下面的值 。
- 每个节点的分支数为k
- 每层的右侧标出当前层中所有节点的和。
- 将所有层总的操作次数相加。

归并排序 (*MS*) 的递归树



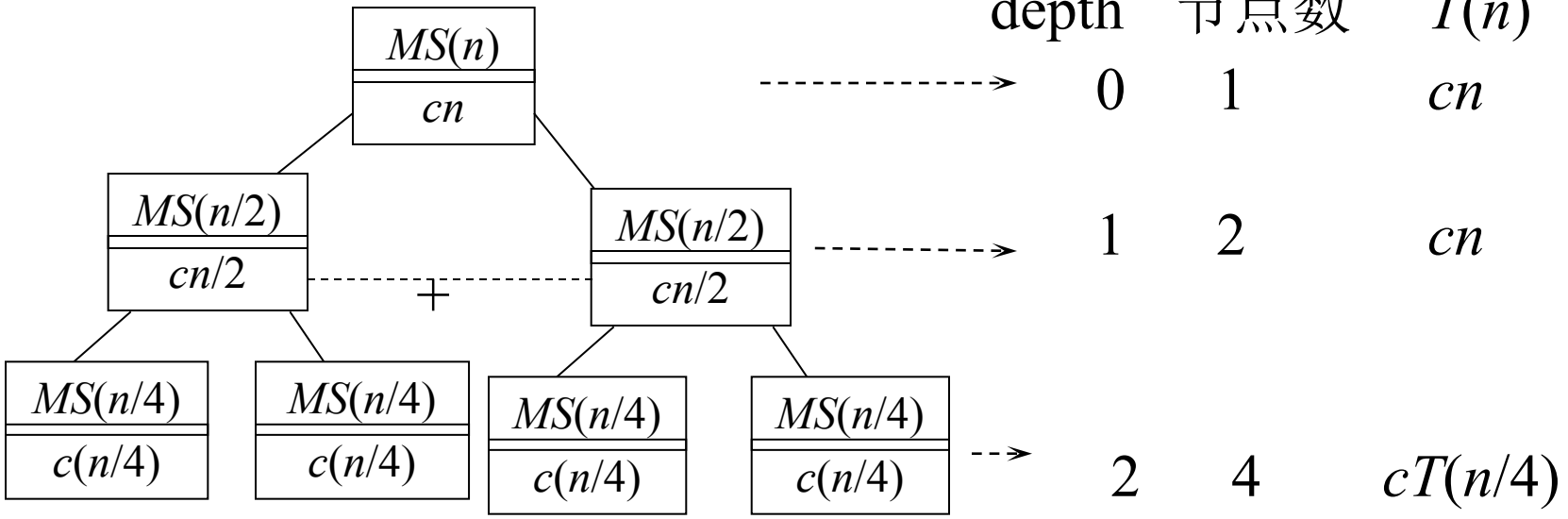
- 初始的，递归树只有一个结点，包含调用 $MS(n)$, 最坏情况下的合并代价是 cn.
- 当展开计算，该节点变成一颗子树:
  - 子树的根结点包含调用 $MS(n)$, 和非递归的操作 cn.
  - 两个孩子结点各包含一个递归调用 $MS(n/2)$, 最坏情况下的合并代价是 $c(n/2)$。

归并排序第一层展开

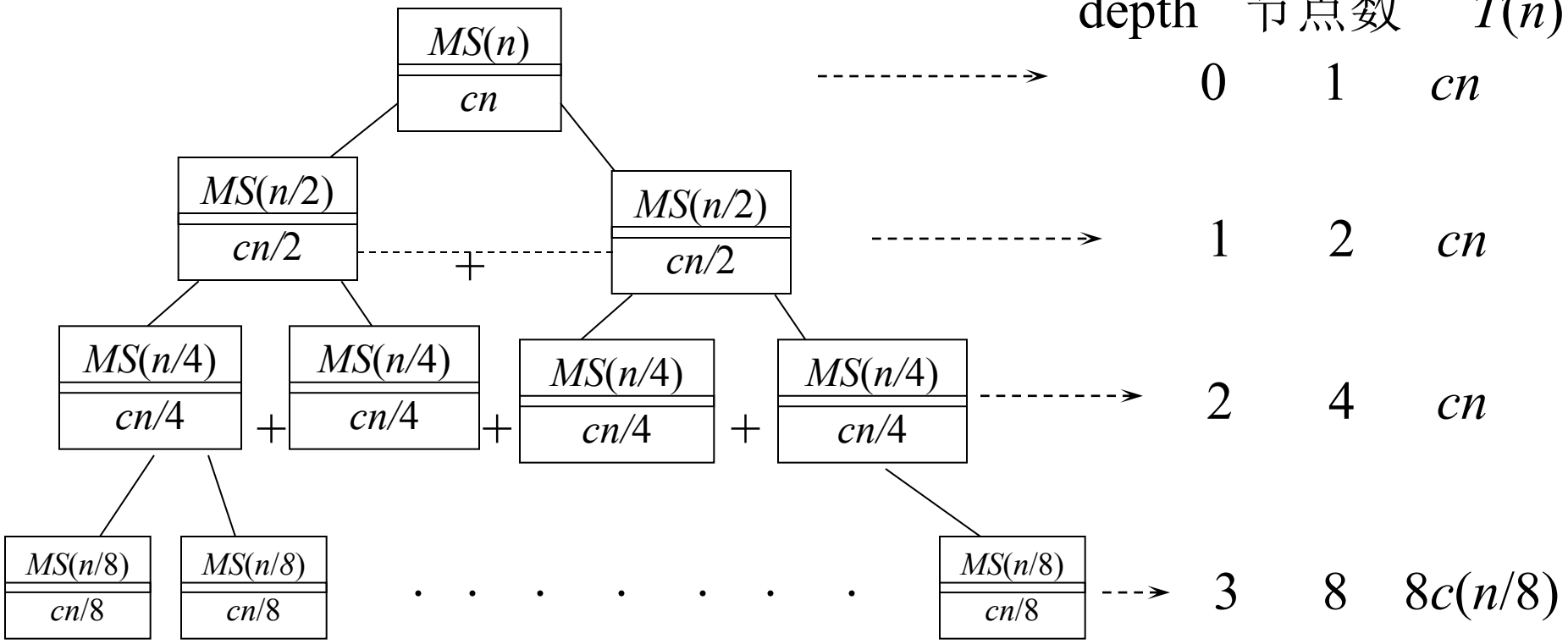


depth	节点数	$T(n)$
0	1	cn
1	2	$2c(n/2)$

第二层展开


MS(n)
cn
MS(n/2)
cn/2
MS(n/2)
cn/2
MS(n/4)
c(n/4)
MS(n/4)
c(n/4)
MS(n/4)
c(n/4)
MS(n/4)
c(n/4)
depth
节点数
T(n)
0
1
cn
1
2
cn
2
4
cT(n/4)

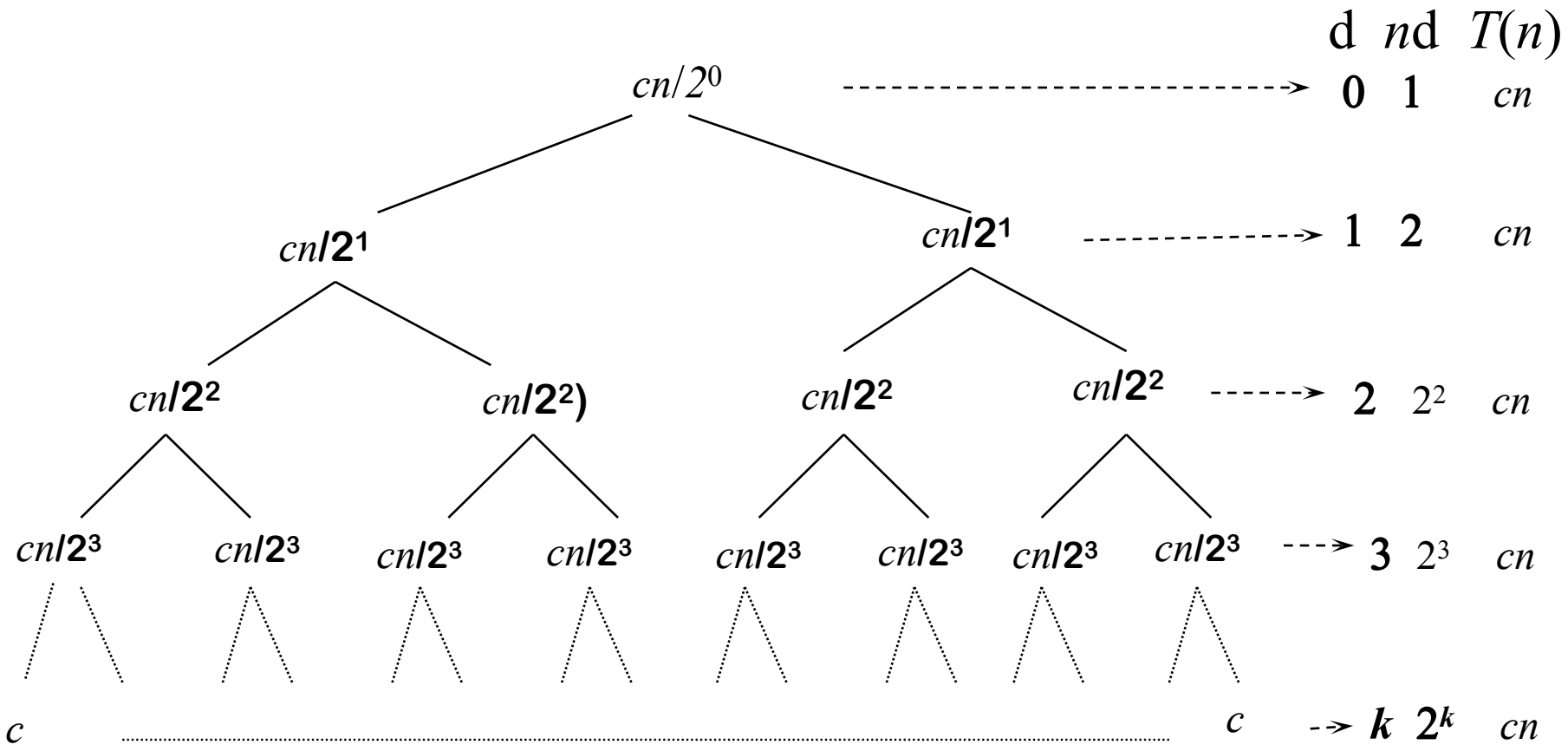
第三层展开


MS(n)
cn
MS(n/2)
cn/2
MS(n/2)
cn/2
MS(n/4)
cn/4
MS(n/4)
cn/4
MS(n/4)
cn/4
MS(n/4)
cn/4
MS(n/8)
cn/8
MS(n/8)
cn/8
MS(n/8)
cn/8
depth 节点数 T(n)
0 1 cn
1 2 cn
2 4 cn
3 8 8c(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 $\Theta(1)$

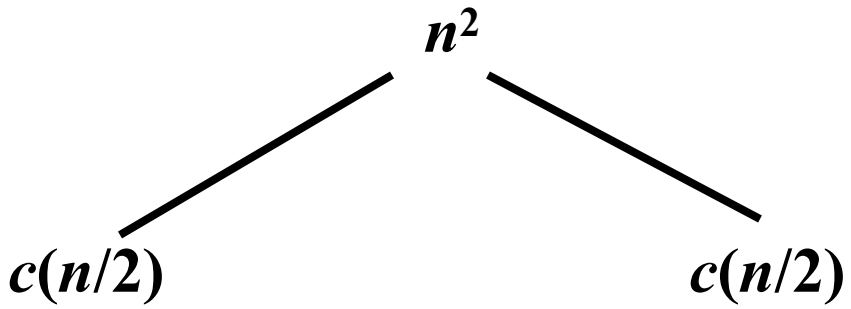
完整的递归树



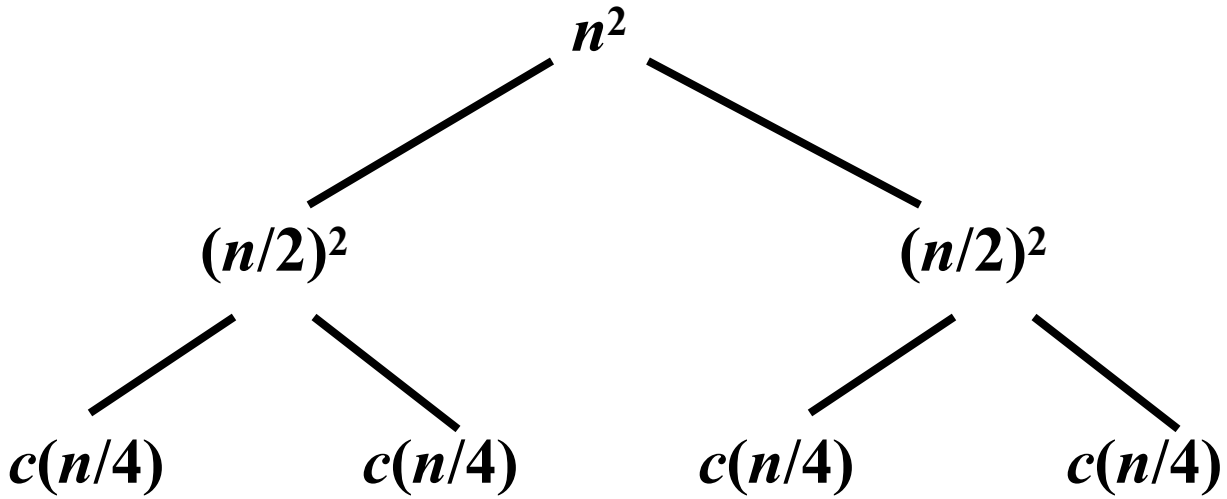
$$T(n) = (k+1)(cn) = (\lg n + 1)(cn) = \Theta(n \lg n)$$

举例: $T(n) = 2T(n/2) + n^2$

第一层展开:

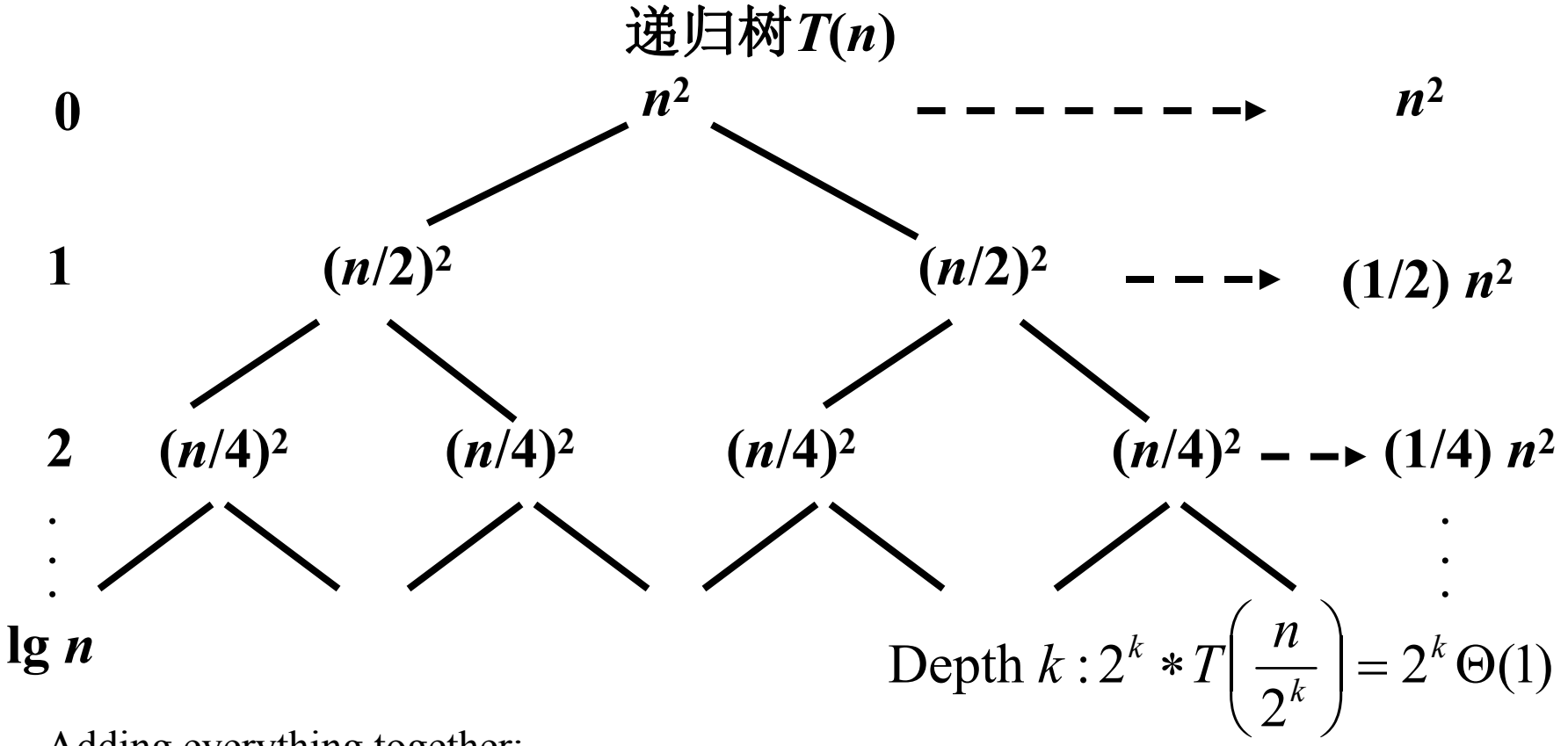


第二层展开:



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

举例: $T(n) = 2T(n/2) + n^2$

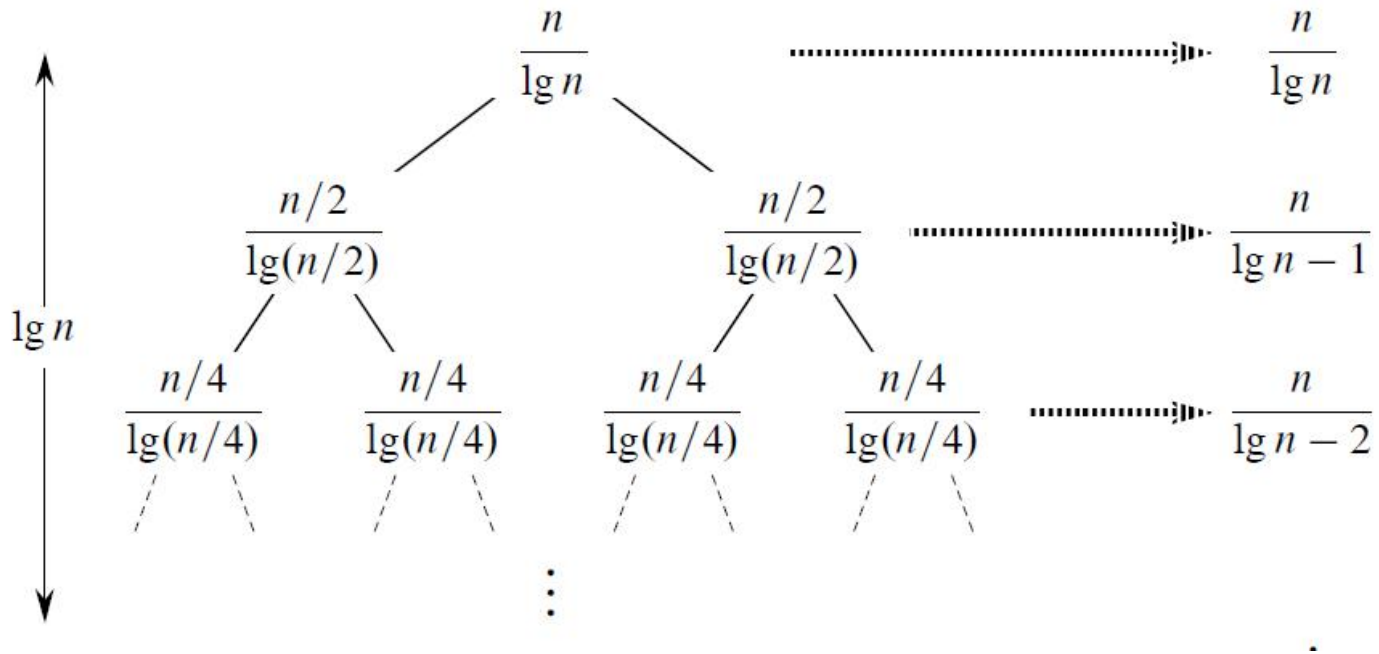


Adding everything together:

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

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

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

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

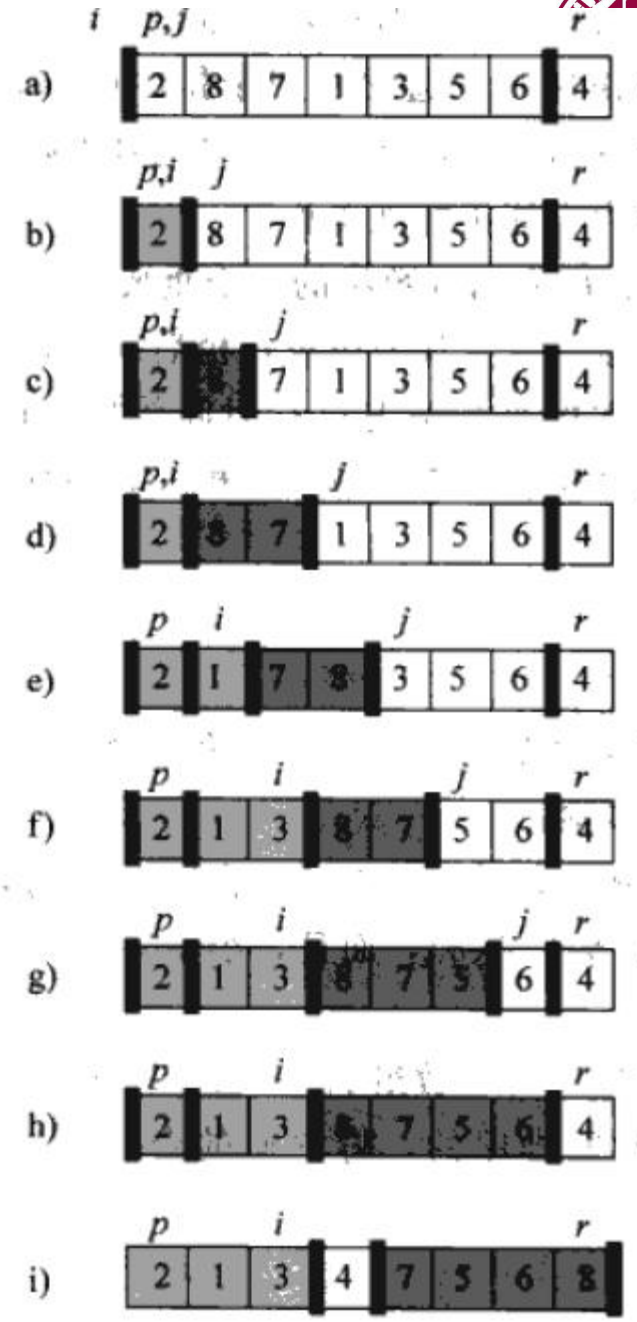
➔ $T(n)=\Theta(n\lg\lg n)$

调和级数:

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

```
QUICKSORT(A, p, r)
1  if p<r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q−1)
4          QUICKSORT(A, q+1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p−1
3  for j ← p to r−1
4       do if A[j]≤x
5             then i ← i+1
6                  exchange A[i]↔A[j]
7  exchange A[i+1]↔A[r]
8  return i+1
```



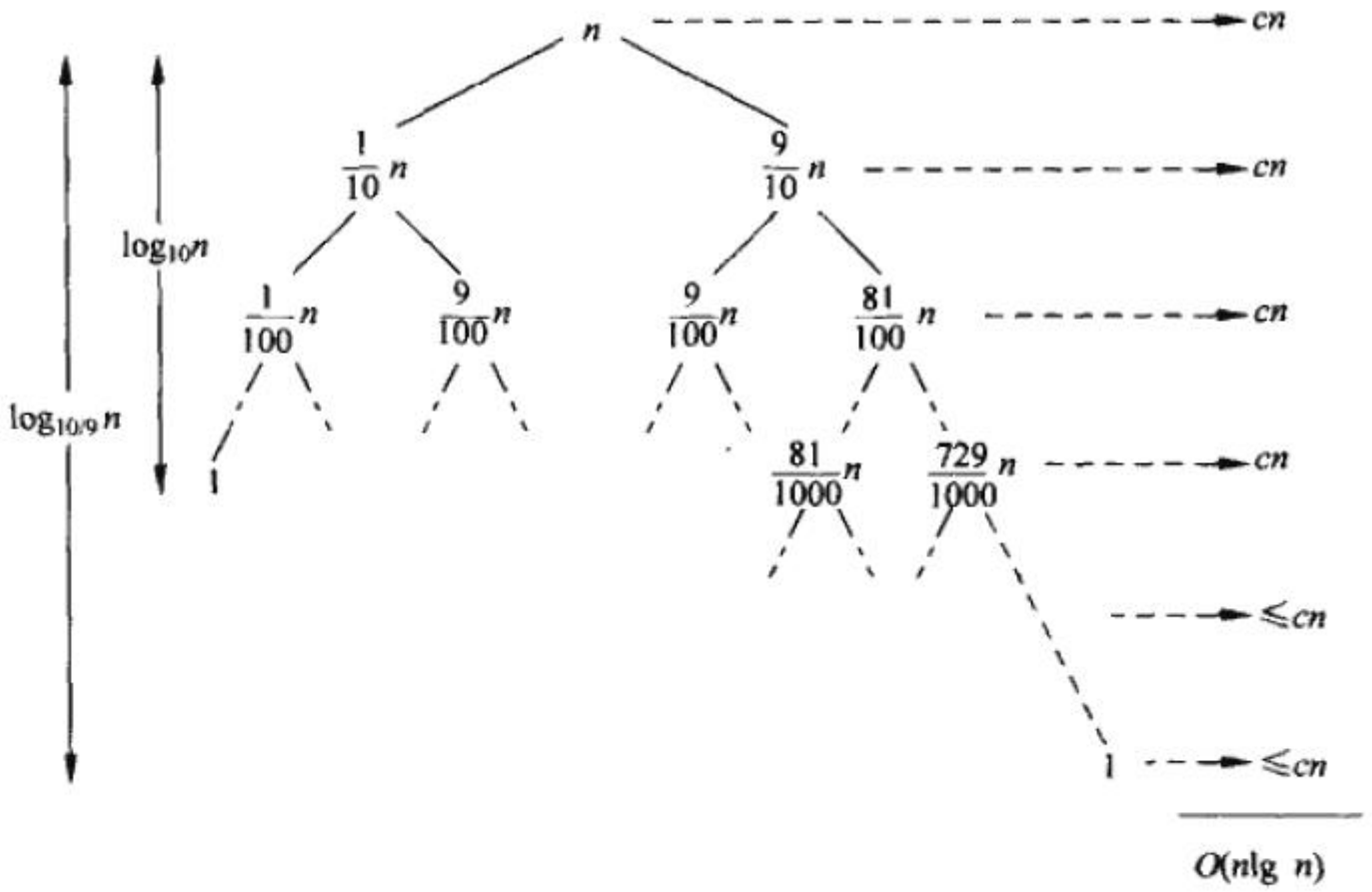


图 7-4　QUICKSORT 的一棵递归树，其中 PARTITION 总是产生 9：1 的划分，总的运行时间为 $O(n \lg n)$。各结点中示出了子问题的规模，每一层的代价在右边显示。每一层的代价包含了 $\Theta(n)$ 项中所隐含的常数 c

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

$$T(n) \leqslant 2T(n/2) + \Theta(n)$$

例如，假设划分过程总是产生 9∶1 的划分，乍一看这种划分很不平衡，这时，快速排序运行时间的递归式为

$$T(n) \leqslant T(9n/10) + T(n/10) + cn$$

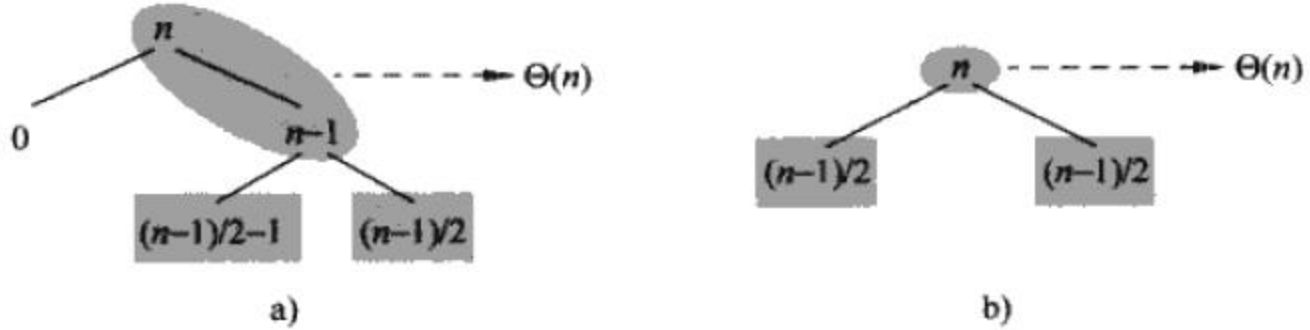


图 7-5　a)快速排序的递归树中的两层。树根处划分的代价为 n，且得到是“很差的”划分：两个子数组的大小分别为 0 和 $n-1$。对大小为 $n-1$ 的子数组的划分代价为 $n-1$，产生一个“较好的”划分：两个子数组的大小分别为 $(n-1)/2-1$ 和 $(n-1)/2$；b)一棵非常平衡的递归树中的一层。在它的两个部分中，子问题的划分代价以椭圆形阴影示出，为 $\Theta(n)$。但是，a)图中矩形阴影示出的、待解决的子问题并不大于 b)图中对应的、待解决的子问题

做，但是，如果采用一种不同的、称为随机取样(random sampling)的随机化技术的话，可以使得分析更加简单。在这种方法中，不是始终采用 $A[r]$作为主元，而是从子数组 $A[p..r]$中随机选择一个元素，即将 $A[r]$与从 $A[p..r]$中随机选出的一个元素交换。在这一修改中，我们是从 p，…，r这一范围中随机取样的，这么做确保了在子数组的 $r-p+1$ 个元素中，主元元素 $x=A[r]$等可能地取其中的任何一个。因为主元元素是随机选择的，我们期望在平均情况下，对输入数组的划分能够比较对称。

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r]↔A[i]
3  return PARTITION(A, p, r)
```

新的快速排序过程不再调用 PARTITION，而是调用 RANDOMIZED-PARTITION。

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p<r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q-1)
4          RANDOMIZED-QUICKSORT(A, q+1, r)
```

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

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:

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

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

  - $f(n) = n \lg^{-1} n$ 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。

  - $f(n) = n \lg^{-1} n$ is not in $\Theta(n^{\log b^a} \lg^k n)$ for any $k \geq 0$

    ➔ 主定理 Case 2 不适用。

  - $f(n) = n \lg^{-1} n$ 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:

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

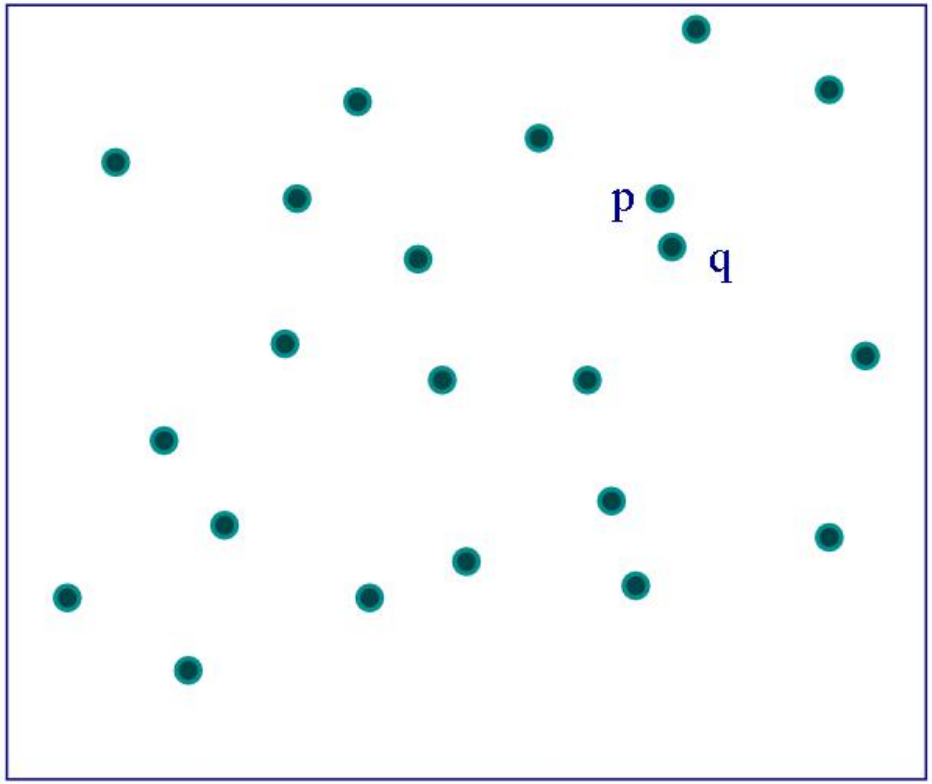
解 $S(m)$: $\in \Theta(m \lg m)$

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

二维最近对问题

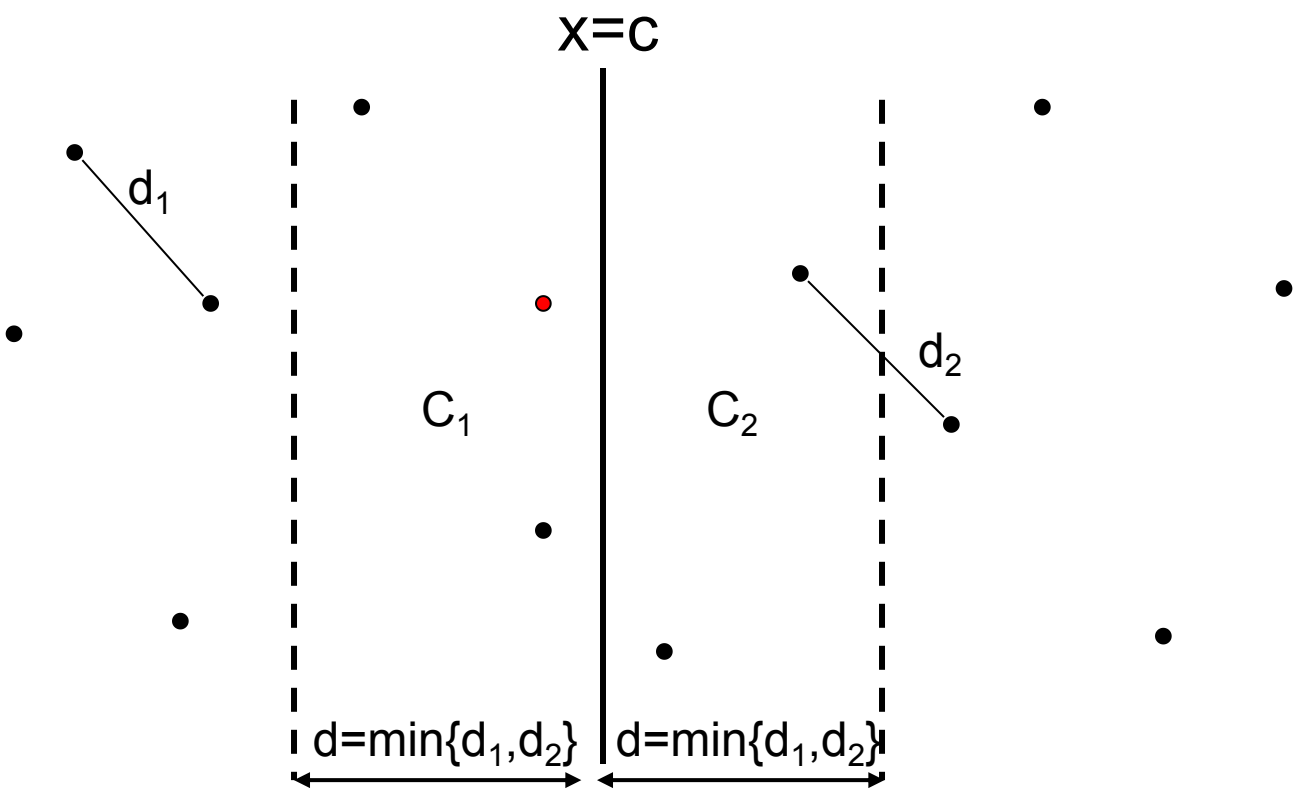
- $P_1(x_1,y_1),\ldots,P_n=(x_n,y_n)$ 是平面上n个点构成的集合S，假设$n=2^k$，最近点对问题要求找出距离最近的两个点

- 最近点对问题是许多算法的基本步骤



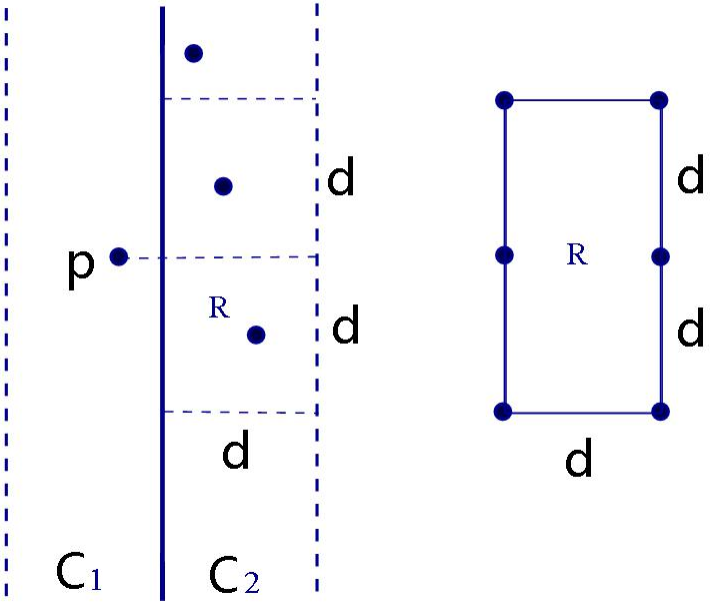
二维最近点对问题

- 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(nlogn)

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

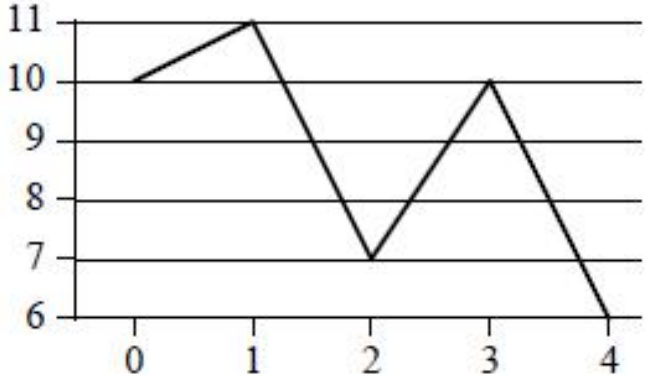
- 考虑下面的情景:
  - 一支股票连续n天的交易价格。
  - 什么时间该买入? 什么时间该卖出?
- 如何将这个问题转换成最大子数组问题?

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 -1天买入。
  - 第 j 天卖出。

最大子数组问题应用

- 一支股票连续n天的交易价格:



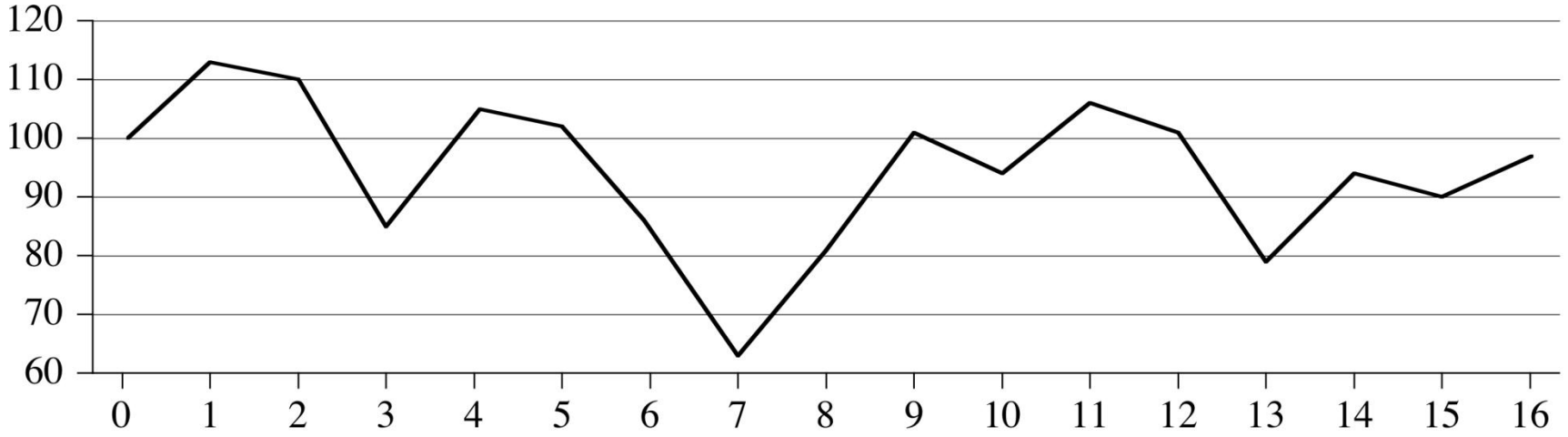
Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

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?

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

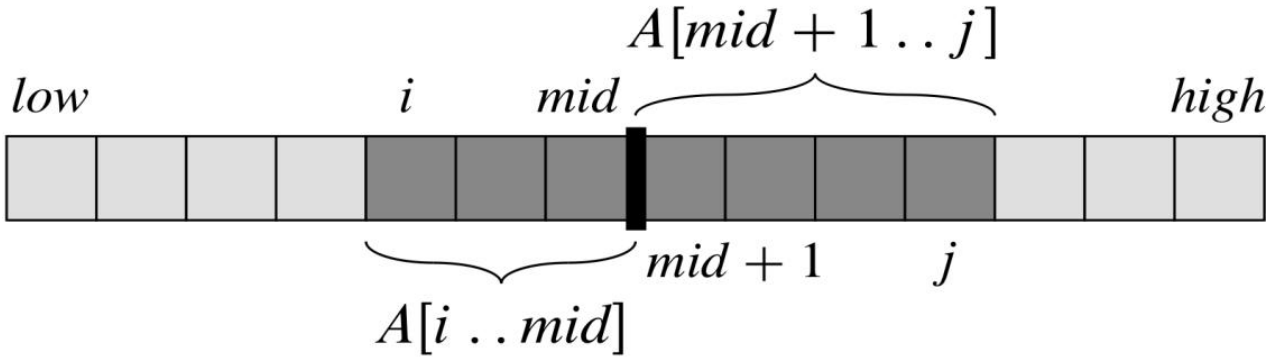
- 然后计算每个子数组的和
  - 对于每一个子数组，需要做多少次加法?
    - 取决于子数组的大小: 从0 到 $n-1$.
    - 至少是 $\Omega(1)$.
- 最后找出最大的和: $\Theta(n^2)$.
- 蛮力法需要 $\Omega(n^2)$ 时间.
- 如何算得更快?

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)。

找跨越中间位置的最大子数组

- 这是一个关键的新问题。
- 它不是原问题的一个小规模实例
  - 附加限制：子数组必须跨越中间位置。
- 这个问题可以用$\Theta(n)$时间解决。
  - 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$low \le i \le mid < j \le high$.
  - 因此，只需要找两种形式的最大子数组 $A[i \,..\, mid]$ 和 $A[mid+1 \,..\, j]$, 然后把它们合并。



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.

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)

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

- 传统方法：$O(n^3)$
- 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：

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

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$n=2^k$，$M(n)=7M(n/2)=7^2M(n/2^2)\cdots\cdots$
    $=7^kM(1)=7^k$
  - $M(n)=7^{\log_2 n}=n^{\log_2 7}\approx n^{2.807}$

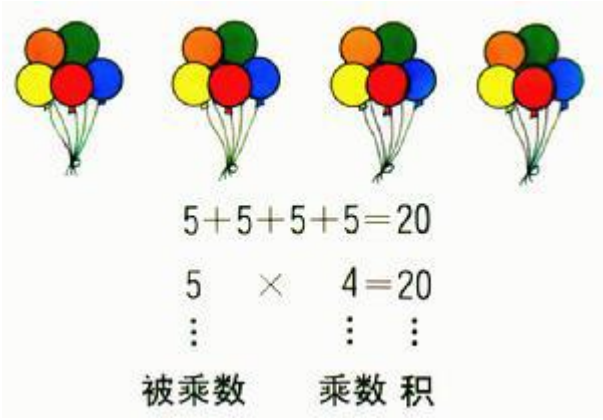
➢Hopcroft和Kerr已经证明(1971)，计算2个 2 × 2 矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究 3 × 3 或 5 × 5 矩阵的更好算法。

➢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。

大整数乘法

- 是指将相同的数加法起来的快捷方式。其运算结果称为积。



23*14=?
23+23+23+23+…

大整数乘法

$23 = 2 \cdot 10^1 + 3 \cdot 10^0$ 和 $14 = 1 \cdot 10^1 + 4 \cdot 10^0$

现在我们把它们相乘：

$$23 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2 * 1)10^2 + (3 * 1 + 2 * 4)10^1 + (3 * 4)10^0$$

- 思路：任意两个数字相乘，然后把乘的值加到最后的结果上面去

抽象

- A: a1a0=a1*10+a0
- B: b1b0=b1*10+b0
- C=A*B= (a1*10+a0) (b1*10+b0)=a1*b1*10^2+(a0*b1+a1*b0)*10+a0 *b0
-
- 再抽象一步
- A: a_{n-1} …. a_0
- B: bn-1 … b0
- C: m2n-1….m0
- C=an-1*bn-1*10^(2n-2)+(an-1*bn-2+an-2+bn-1) 10^(2n-1)…(a1*b0+a0*b1)*10+a0*b0

- 现在ai×aj，这个值要加到M上，问题是加到M的哪一位？
- 加到第i+j位
- 也即M=M+ai*aj*10^(i+j)
-
- A和B都是n位，那么M最多可以有多少位？
- M最多可以有2n+1位

算法

- Multiply(A,B,M)
- 输入：A，B，n位的数组
- 输出：C，2n位的数组
-
- c=0
- for (i=0;i<n;i++)
- for(j=0;j<n;j++)
- x= C[i+j]+ai*aj
- C[i+j]= x mod 10
- C[i+j+1]= C[i+j+1]+x / 10

- 比起开始的思路，这里的算法有几点不同
- 在思路中，我们说一个n位的数，而在算法中，我们使用了数组来表示这个数
- 在思路中，我们直接把相乘的结构加到了C上，而且假定进位可以直接实现。在算法中，因为使用了数组，进位操作必须自己实现。

分析

- Multiply(A,B,M)
- 输入：A，B，n位的数组
- 输出：C，2n+1位的数组
-
- M=0
- for (i=0;i<n;i++)
- for(j=0;j<n;j++)
- x= C[i+j]+ai*aj
- C[i+j]= x mod 10
- C[i+j+1]= C[i+j+1]+x / 10

基本操作：乘法　每次循环，做1次乘法，1次求余，1次出发，2次加法，另外还有求下表的6次加法

次数： 3n^2乘法，8 n^2加法

算法复杂度：O（n^2）

- 注意1.3.1.4：
- （1）我们断言，相对于乘法来说，加法所花的时间可以忽略，大家可以验证这一点，比如做1万、10万、100万、1000万个加法，然后再做同样次数个乘法，比较一下两者的时间。
- （2）我们断言，做一次乘法和做1次除法的时间差不多，同样试着验证一下
- （3）我们断言，做一次乘法和做3次乘法的时间差不多，请验证一下
- （4）最后的算法复杂度是按照最耗时的操作来计算的，忽略次要的选项

实现

第一种实现方式（C++）：

```
#include<iostream>
#include<stdlib.h>
#include <time.h>
#include<ctime>
using namespace std;
void multiply(int *a,int *b,int n)
{

    int *c=new int[2*n];
    int x=0;
    for(int i=0;i<2*n;i++)
    {
        c[i]=0;
    }
    for(i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            x=c[i+j]+a[i]*b[j];
            c[i+j]=x%10;
            c[i+j+1]=c[i+j+1]+x/10;
        }
    }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define N 10000

void multiply(int *s1,int *s2,int *c)
{
    int i,j;
    //数组初始化
    for(i = 0; i < 2 * N; ++i)
        *(c + i) = 0;

    //相乘
    for(i = 0; i < N; ++i)
        for(j = 0; j < N; ++j)
            *(c + i + j) += *(s1 + i) * *(s2 + j);
    //统一处理进位
    for(i = 0; i < 2 * N - 1; ++i)
    {
        *(c + i + 1) += *(c + i)/10;//将十位上的数向前进位，并加上原来这个位上的数
        *(c + i) = *(c + i)%10;//将剩余的数存原来的位置上
    }
}
```

- 相对于算法，算法的C语言实现有几个区别
- include引入相应的程序包
- 很多c或者C++的语法，比如循环for，比如c+=a，比如传递参数采用指针而不是数组
- 为了更好地理解代码，一般要使用注释
- 可见，一般人能够理解的，其实是伪代码级别

分治法（1）：思路

- $A=a_1 10^{n/2}+a_0$　　$B=b_1 10^{n/2}+b0$

  直接计算

$$
\begin{aligned}
c = a \times b &= (a_1 10^{n/2} + a_0) \times (b_1 10^{n/2} + b_0) \\
&= (a_1 \times b_1) 10^n + (a_1 \times b_0 + a_0 \times b_1) 10^{n/2} + (a_0 \times b_0) \\
&= c_2 10^n + c_1 10^{n/2} + c_0
\end{aligned}
$$

- A=a1*10n/2+a0　，B=b1*10n/2+b0
- 直接计算：
- A·B=(a1*10n/2+a0)(b1*10n/2+b0)
- =a1*b1*10n+(a1*b0+a0*b1)10n/2+a0*b0

分治法（1）：伪代码

- Mutiply(X,Y,n)
- If n=1 return X*Y
- A1=X的高n/2位
- A0=X的低n/2位
- B1=Y的高n/2位
- B0=Y的低n/2位
- P= multiply(A1, B1, n/2)
- Q= multiply(A1,B0, n/2)
- R= multiply(A0,B1,n/2)
- S= multiply(A0, B0, n/2)
- C=P<<n+(Q+R)<<n/2+S
- ret

计算乘法的递归公式（不是计算整个算法时间）为：

$C(n)=4C(n/2)$

$C(1)=1$

解得 $C(n) \in O(n2)$

分治法（2）：思路

- $A \cdot B=(a1 10^{n/2}+a0)(b1 10^{n/2}+b0)$
- $=a1b1 10^{n}+(a1b0+a0b1)10^{n/2}+a0b0$
- $=a1b1 10^{n}+[(a1+a0)(b1+b0)-a1b1-a0b0)]10^{n/2}+a0b0$
-
- 验证：
- $[(a1+a2)(b1+b2)-a1b1-a2b2)]=(a1b2+a2b1)$

分治法（2）：伪代码

- Mutiply(X,Y,n)
- If n=1 return X*Y
- A1=X的高n/2位
- A0=X的低n/2位
- B1=Y的高n/2位
- B0=Y的低n/2位
- P= multiply(A1, B1, n/2)
- A2=A1+A0
- B2 =B1+B0
- Q= multiply(A2,B2 , n/2)
- S= multiply(A0, B0, n/2)
- C=P<<n+(Q-P-S)<<n/2+S

这个算法有什么问题？
没有考虑进位
没有给出大整数加法
Q<<n这种移位操作时基于二进制进行的，而不是十进制
n/2如果不是整数怎么办？
最重要的，递归会非常消耗时间

分治法（2）：分析

$$c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$$

这种方法需要3次n/2位的乘法及一些加减法。

该算法会做多少次位乘呢？因为 n 位数的乘法需要对 $n/2$ 位数做三次乘法运算，乘法次数 $M(n)$的递推式将会是：

$$\text{当 } n>1 \text{ 时，} M(n)=3M(n/2)，\ M(1)=1$$

当 $n=2^k$ 时，我们可以用反向替换法对它求解：

$$M(2^k)=3M(2^{k-1})=3[3M(2^{k-2})]=3^2M(2^{k-2})$$
$$=\cdots=3^iM(2^{k-i})=\cdots=3^kM(2^{k-k})=3^k$$

因为 $k=\log_2 n$，

$$M(n)=3^{\log_2 n}=n^{\log_2 3}\approx n^{1.585}$$

(在最后一步中，我们利用了对数的一个特性：$a^{\log_b c}=c^{\log_b a}$。)

我们应当知道，对于不是很大的整数，该算法的运行时间很可能比经典算法长。Brassard 和 Bratley([Bra96]，pp.70-71)报告说，他们的实验显示，从大于 600 位的整数开始，分治算法的性能超越了笔算算法的性能。如果我们使用类似 Java，C++和 Smalltalk 这样的面向对象语言，会发现这些语言专门为处理大整数提供了一些类。

分治法2：例子

- X=3141　　A=31　　B=41　　A-B=-10
- Y=5327　　C=53　　D=27　　D-C=-26
- AC=
- BD
- (A-B)(D-C)=

- $A=31 \quad A_1=3 \quad B_1=1 \quad A_1-B_1=2$
- $C=53 \quad C_1=5 \quad D_1=3 \quad D_1-C_1=-2$
- $A_1C_1=15 \quad B_1D_1=3 \quad (A_1-B_1)(D_1-C_1)=-4$
- $AC=1500+(15+3-4)10+3=1643$
- $B=41 \quad A_2=4 \quad B_2=1 \quad A_2-B_2=3$
- $D=27 \quad C_2=2 \quad D_2=7 \quad D_2-C_2=5$
- $A_2C_2=8 \quad B_2D_2=7 \quad (A_2-B_2)(D_2-C_2)=15$
- $BD=800+(8+7+15)10+7=1107$
- $|A-B|=10 \quad A_3=1 \quad B_3=0 \quad A_3-B_3=1$
- $|D-C|=26 \quad C_3=2 \quad D_3=6 \quad D_3-C_3=4$
- $A_3C_3=2 \quad B_3D_3=0 \quad (A_3-B_3)(D_3-C_3)=4$
- $(A-B)(D-C)=200+(2+0+4)10+0=260$

- X=3141　　A=31　　B=41　　A-B=-10
- Y=5327　　C=53　　D=27　　D-C=-26
- AC=(1643)'
- BD=(1107)'
- (A-B)(D-C)=(260)'
- $XY=(1643)'10^4+[(1643)'+(260)'+(1107)']10^2+(1107)'$
- =(16732107)'

俄式乘法（减治法）

- 如果n是偶数
- 如果n是奇数

$$n \times m = \frac{n}{2} \times 2m$$

$$n \times m = \frac{n-1}{2} \times 2m + m$$

-

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1 040	
1	2 080	(+1040)
	2 080	+(130 + 1040) = 3 250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1 040	1 040
1	2 080	2 080
		3 250

(b)

图 5.14　用俄式乘法计算 50×65

该算法的效率是多少？

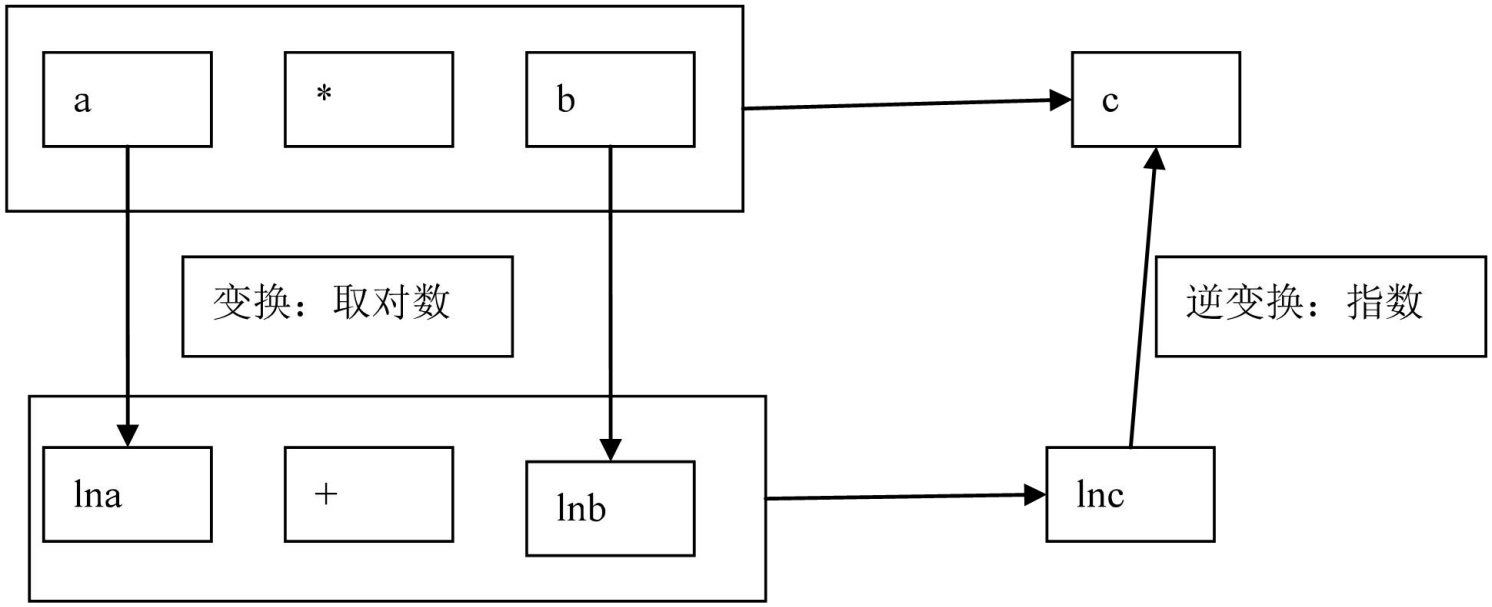
俄式乘法：算法

- Multiply(A,B,M)
- 输入：A，B，n位的数组，基数是二进制
- 输出：D，2n+1位的数组，基数是二进制
- C=B
- D=0
- for (i=0;i<n;i++)
- if A[i]=0 then
- C向右移动一位
- Else
- D=D+C
- C向右移动一位
- D=D+C
- Return D

问题：这个算法只有一层循环，是不是复杂度是O（n）？
不是，C向右移动1位，如果C有2n位，那么移动一位的代价是O（n）

问题：如果直接在C后面添加0，行不行？或者用链表的方式实现，会不会提高效率？

快速傅里叶变换（变治法）

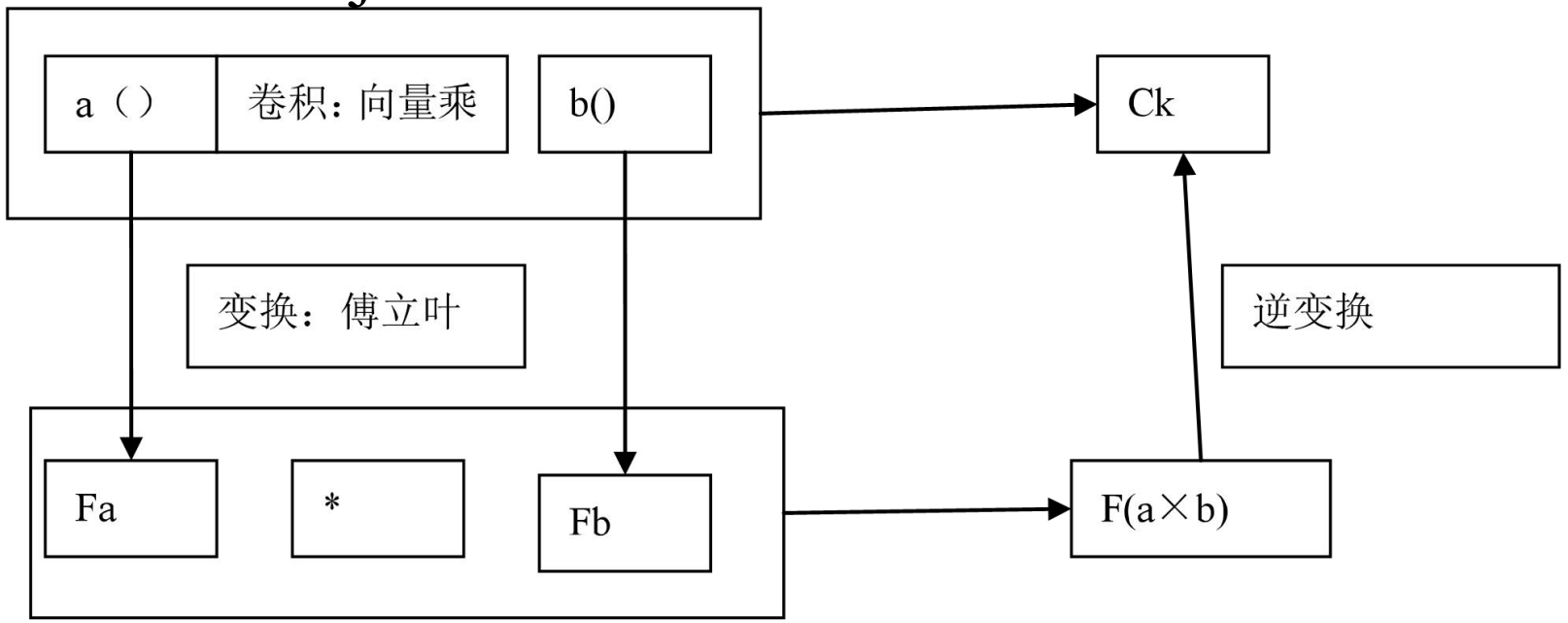


例子：a=456, b=123

- 思路是有趣的，问题：
- ln和计算指数的时间更多。解决方法：计算对数和指数的值可以编制成表。但是编制表之后又要解决精度问题
- 当a和b很大的时候，只能以数组表示，这时候应该如何计算对数？

- $c_k = a_0 x b_k + a_1 x b_{k-1} + \ldots + a_{k-2} x b_2 + a_{k-1} x b_1 + a_k x b_0 + a_{k+1} x b_{-1} + \ldots + a_{N-2} x b_{-(N-2-k)} + a_{N-1} x b_{-(N-1-k)}$

- Ck=（ai，bj）



- 假设我们要计算以下两个 N 位数字的乘积：
- $a = (a_{N-1}a_{N-2}...a_1a_0)_{10} = a_{N-1}x10^{N-1} + a_{N-2}x10^{N-2} + ... + a_1x10^1 + a_0x10^0$
- $b = (b_{N-1}b_{N-2}...b_1b_0)_{10} = b_{N-1}x10^{N-1} + b_{N-2}x10^{N-2} + ... + b_1x10^1 + b_0x10^0$
- 将上面两个式子相乘，得到以下公式 (共 2N - 1 项)：
- $c = a \times b = c_{2N-2}x10^{2N-2} + c_{2N-3}x10^{2N-3} + ... + c_1x10^1 + c_0x10^0$
- 非常容易验证，上式中的 c_k ($0 \le k \le 2N-2$) 满足以下公式：
- $c_k = a_0xb_k + a_1xb_{k-1} + ... + a_{k-2}xb_2 + a_{k-1}xb_1 + a_kxb_0 + a_{k+1}xb_{-1} + ... + a_{N-2}xb_{-(N-2-k)} + a_{N-1}xb_{-(N-1-k)}$
- 上式共有 N 项，a_i 和 b_j 的下标 i 和 j 满足 $i + j = k$。若不满足 $0 \le i, j \le N-1$ 时，则令 $a_i = b_j = 0$。
- Ck=（ai，bj）

我们可以按照以下步骤来计算大整数乘法：

分别求出向量 {ai} 和向量 {bj} 的离散傅里叶变换 {Ai} 和 {Bj}。

将 {Ai} 和 {Bj} 逐项相乘得到向量 {Ck}。

对 {Ck} 求离散傅里叶逆变换，得到的向量 {ck} 就是向量 {ai} 和向量 {bj} 的卷积。
对的向量 {ck} 进行适当的进位就得到了大整数 a 和 b 的乘积 c。

关键在于：直接进行离散傅里叶变换的计算复杂度是 $O(N^2)$。

快速傅里叶变换可以计算出与直接计算相同的结果，但只需要 O(N logN) 的计算复杂

由于快速傅里叶变换是采用了浮点运算，因此我们需要足够的精度，
以使在出现舍入误差时，结果中每个组成部分的准确整数值仍是可辨认的。

- 更多分析，参见:
- Knuth, The art of computer programming
- 第3卷：半数值算法
- 4.3.3 乘法能有多快

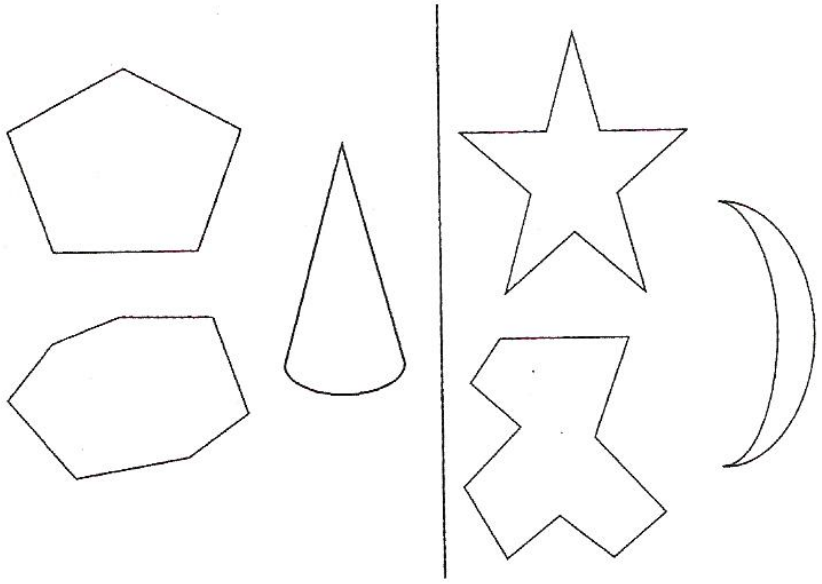
- Pp. 267

凸包问题(Convex Hulls Problem)

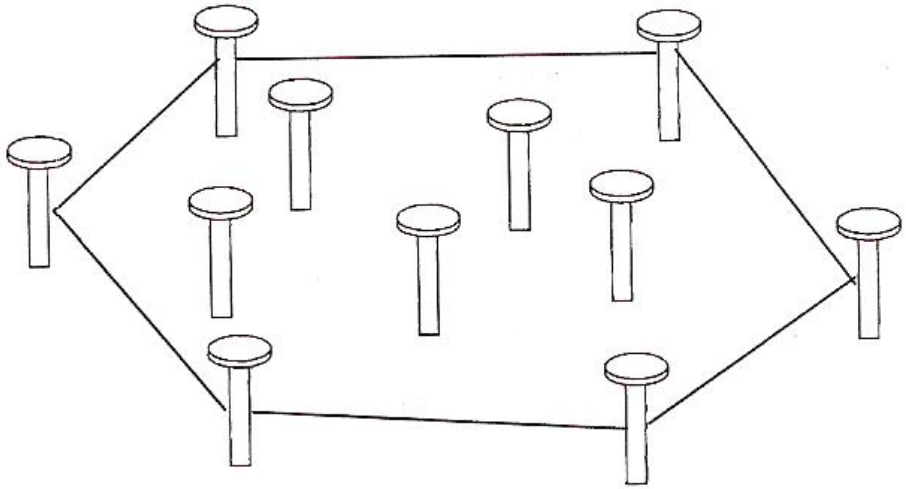
Algorithm	Speed	Discovered By
Brute Force	$O(n^3)$	[Anon, the dark ages]
Gift Wrapping	O(nh)	[Chand & Kapur, 1970]
Graham Scan	O(n log n)	[Graham, 1972]
Jarvis March	O(nh)	[Jarvis, 1973]
QuickHull	O(nh)	[Eddy, 1977], [Bykat, 1978]
Divide-and-Conquer	O(n log n)	[Preparata & Hong, 1977]
Monotone Chain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

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。
- 定义 一个点集合S的凸包是包含S的最小凸集合。



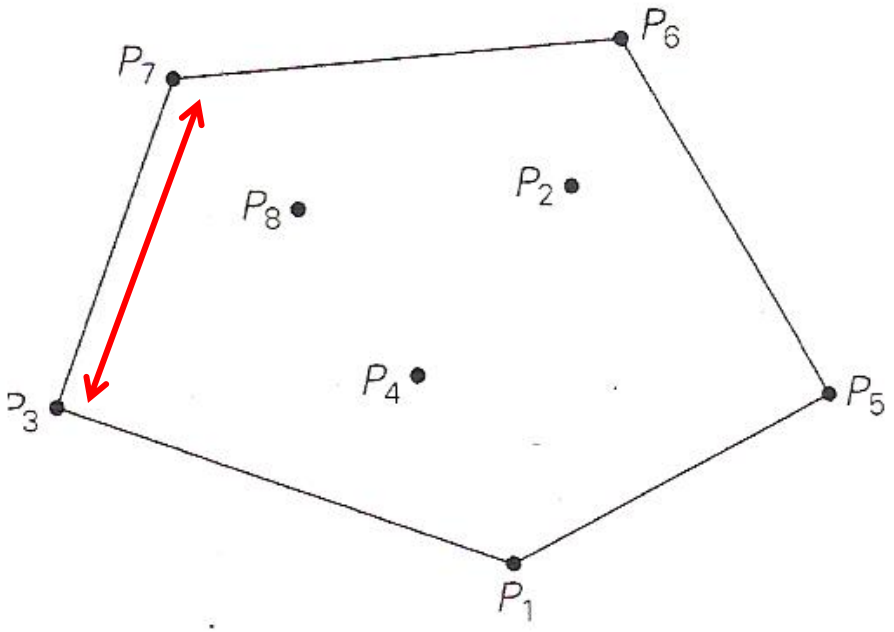
(a)　　(b)



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：对于任何一集合中的点为端点的线段来说，它们不是这种线段的

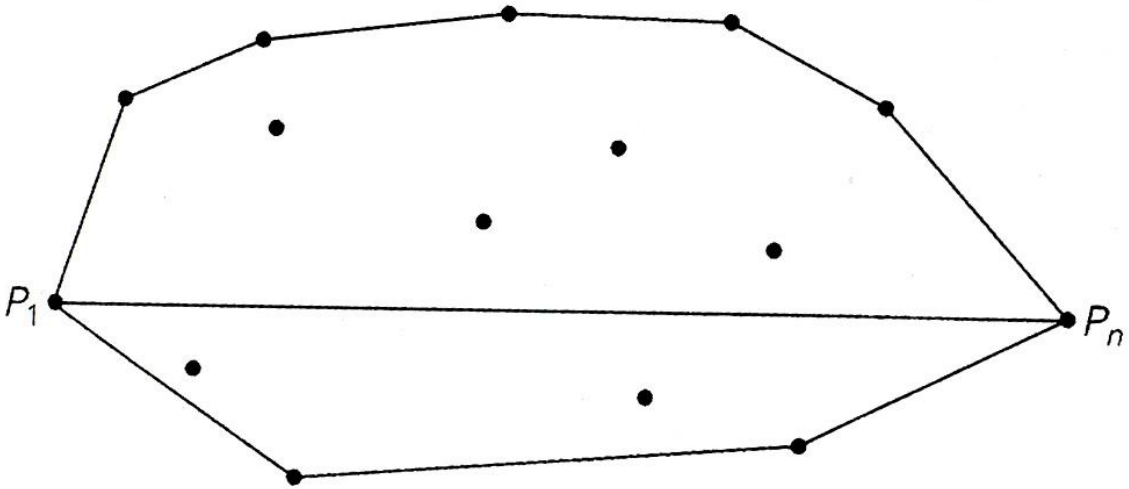
对于一个n个点集合中的两个点P_i和P_j，当且仅当该集合中的其他点都位于穿过这两点的直线的同一边时它们的连线是该集合凸包边界的一部分。对每一对点都做一遍检验之后，满足条件的线段构成了该凸包的边界。



时间效率: $\mathbf{O}(n^3)$

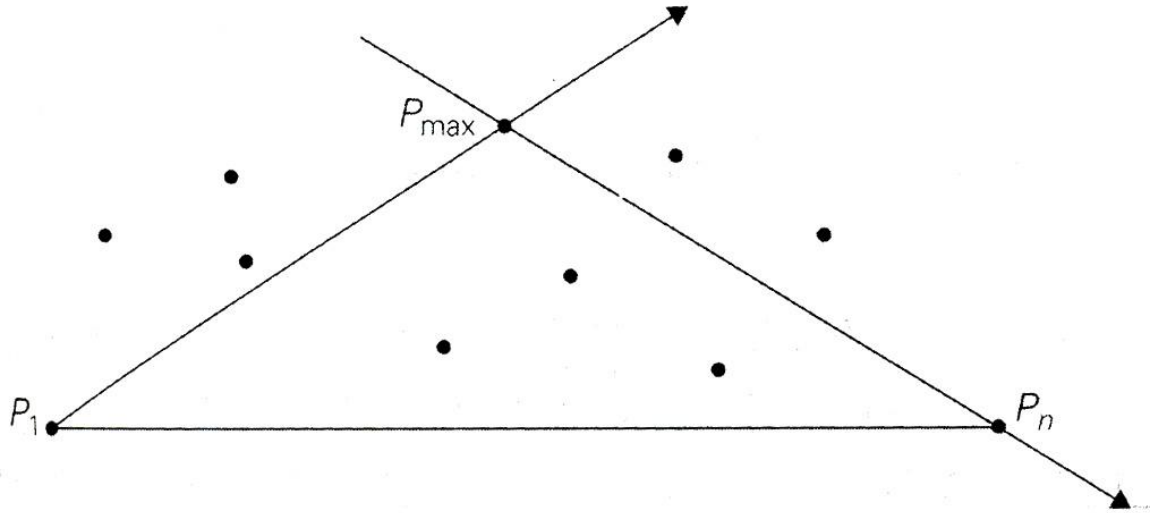
凸包问题(Convex Hulls Problem)

- 假设P1=(x_1,x_2),…,P(x_n,y_n)是平面上n>1个点构成的集合S，并按X轴坐标升序排列，按Y轴坐标升序建立连接。
- 则最左面和最右面的点一定是凸包的顶点。



点集合的上包和下包

凸包问题(Convex Hulls Problem)



Pmax是距离直线P_1P_n最远的点

- P_{max} 是上包的顶点；
- 包含在$\triangle P_1P_{max}P_n$之中的点不可能是上包的顶点(因此在后面不必考虑)；
- 同时位于 $\overrightarrow{P_1P_{max}}$ 和 $\overrightarrow{P_{max}P_n}$ 两条直线左边的点是不存在的。

判断点(x_3,y_3)在顶点为(x_1,y_1)和(x_2,y_2)直线的左右

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

凸包问题效率分析

- 最差效率：$\Theta(n^2)$
- 平均效率：$\Theta(n)$

注意：三角形内部的点不要继续判断

作业

- P50 4.3-1,4.3-2
- P53 4.4-1,4.4-2
- P55 4.5-1
- P60 4-1、4-2、4-3