算法设计与分析

Computer Algorithm Design & Analysis

2025.11

王多强

QQ：1097412466

Chapter 4
Divide-and-Conquer

分治策略

分治法的基本思想：

当问题规模比较大而无法直接求解时，就将原始问题分解为几个规模较小的子问题，然后求解这些子问题。在得到子问题的解后，合并子问题的解而得到原始问题的解。

分治法遵循三个基本步骤：

(1) **分　解**（**Divide**）：将原问题分为若干个规模较小、相互独立、但形式和性质与原问题一样的子问题；

(2) **解　决**（**Conquer**）：若子问题规模足够小、可直接求解，则直接求解；否则“**递归**”求解子问题。

(3) **合　并**（**Combine**）：将子问题的解合并成原问题的解。

要点：

（1）核心思想：**分而治之**；

（2）分解出来的子问题的形式和性质与原始问题一样，从而可以**用递归的方法求解子问题**：

如果分解出来的子问题的规模还比较大、还无法直接求解，则递归求解子问题，即将较大子问题再分解为更小的子问题，直到得到的子问题的规模足够小，此时就不用再分解了，而是直接求解。

（3）**自下而上合并子问题的解**：在得到最小子问题的解后，自下而上层层合并子问题的解而得到较大子问题的解，最后得到原始问题的解。

分治算法的实例：归并排序 详见2.3，P16~22

设 $A[1..n]$ 是含有 n 个元素序列。

- **归并排序的基本思路**：

  分解：将 A 一分为二，得到两个子序列 A_1 和 A_2，它们各有 $n/2$ 个元素。

  解决：递归地对 A_1 和 A_2 进行排序，从而得到关于 A_1 和 A_2 的有序子序列 A'_1 和 A'_2。

  合并：合并 A'_1 和 A'_2，得到关于 A 的完整有序序列 A'。

- **归并排序的过程描述：**
  - MERGE-SORT(A, p, r)
  - MERGE(A, p, q, r)

- **归并排序的时间分析：** $T(n) = 2T(n/2) + cn = O(n\log n)$

◆ 其它已学过的分治算法

- **二分查找**：有序表上的查找算法

  其中一个子问题可能有解，进一步求解；

  另一个子问题确定无解，不用进一步求解。

- **快速排序**：基于划分的排序算法

  划分元素将一个序列分成左右两个子序列，

  然后在两个子序列中继续求解。

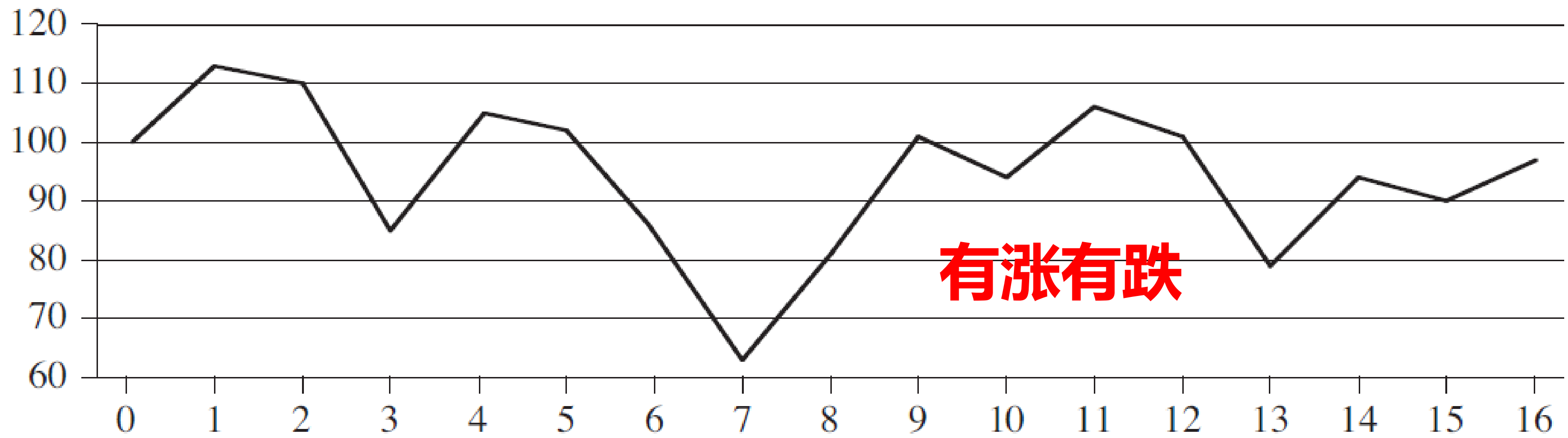
◆ 分治与递归

由于分解出来的**子问题的性质与原问题一样**，所以对子问题的求解实际上是一个递归求解过程。因此，**分治的基本策略就是递归求解。**

- 若子问题的**规模足够小**，就不需要进一步分解（这种情况称为**基本情况**，**base case**)。基本情况下的子问题可以直接求解并返回子问题的答案。
- 若子问题的规模还比较大、需要进一步分解，则递归求解（这种情况称为**递归情况**，**recursive case**)，然后返回当前子问题的解。

4.1 最大子数组问题

- 一个关于炒股的story：



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

相对前一天的收益

每天涨涨跌跌，涨：正收益，迭：负收益，问哪个时间段最赚钱？即从哪天到哪天的收益之和最大？

1、建模求解：

- 求解炒股问题的算法模型：**最大子数组问题**。

  即：已知一个数组 A，在 A 中找一个“**元素之和最大**”的**非空连续子数组** —— 即在数组中找一个这样的区间，该区间内的**所有元素之和**比其它任何区间的元素之和都大（至少是不小于）。

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

相对前一天的收益

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

和最大的子数组

—— 这样的子数组称为 A 的**最大子数组**（***maximum subarray***）。

怎么求一个数组的最大子数组呢？

方法一：暴力求解

枚举A的每对起止下标对应的区间$A[i..j]$，并计算区间和：

$$\sum_{k=i}^{j} a_k,$$

其中**和最大**的区间就是**最大子数组。**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

时间复杂度高：

因为要枚举每对下标，所以暴力求解的时间复杂度是

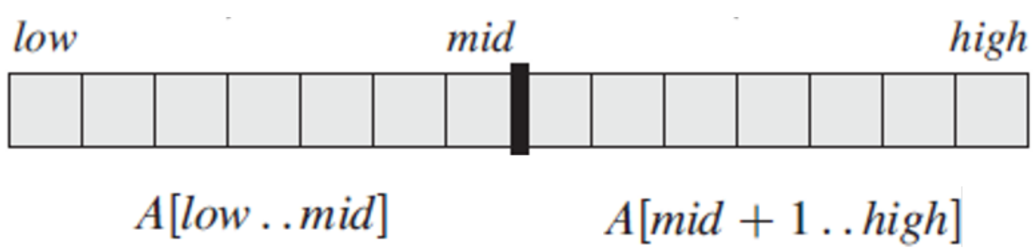
$$\mathbf{n} \cdot \mathrm{C}\binom{n-1}{2} = \Theta(n^3)\text{。}$$

方法二：使用分治策略求解

设当前要寻找数组 A 中区间 $A[low...high]$ 的最大子数组，开始的时候 $low=1$，$high=n$。

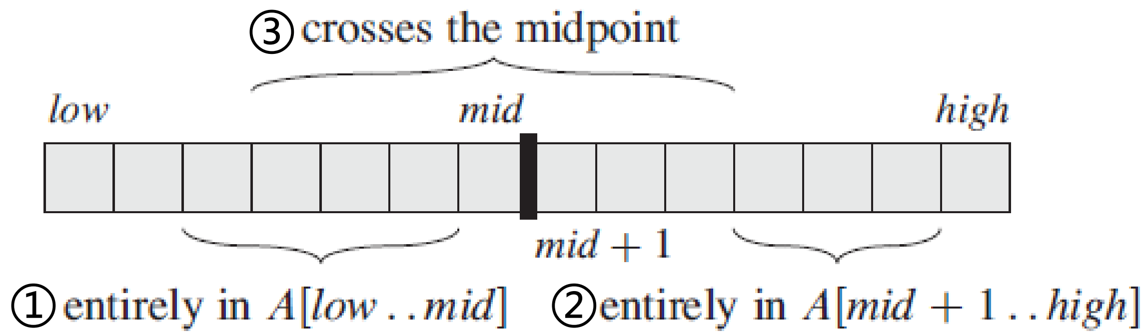
分治法的基本思想是：

首先，计算中间点：$mid = (low+high)/2$，从而将区间 $A[low...high]$ 划分为两个规模大致相等的子区间：



分析：

这使得 $A[low\ldots high]$ 中的任何**连续子数组** $A[i\ldots j]$ $(low \le i \le j \le high)$ 所处的位置将是下面三种情况之一：



① 若$low \le i \le j \le mid$，则 $A[i\ldots j]$ 完全在$A[low\ldots mid]$中；

② 若$mid < i \le j \le high$，则$A[i\ldots j]$ 完全在$A[mid+1\ldots high]$中；

③ 若$low \le i \le mid < j \le high$，则 $A[i\ldots j]$ **跨越中间点**。

因为 $A[low\ldots high]$ 中的**最大子数组**也是一个**连续子数组**，所以最大子数组最终所处的位置也必然是上述三种情况之一：或者在 mid 左侧，或者在 mid 右侧，或者横跨 mid。

2、求最大子数组问题的分治算法

分别求出上述**三种情况**下的最大子数组：即仅在左侧 *A*[*low*...*mid*] 范围内、仅在右侧 *A*[*mid*+1..*high*] 范围内，以及恰好**横跨** *mid* 的最大子数组，然后比较三者**和值的大小**，其中和最大的那个就是整个 *A*[*low*..*high*] 区间的最大子数组。

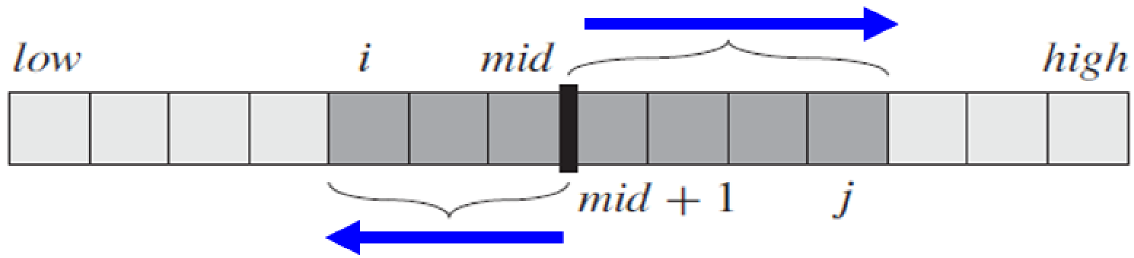
(1) 求仅在 *mid* 左侧和右侧的最大子数组——递归

如果一个算法是求区间 *A*[*low*..*high*] 的最大子数组，则在左侧区间 *A*[*low*..*mid*] 上**递归调用**该算法，即可求得仅在 *mid* 左侧的最大子数组。

同理，在右侧区间 *A*[*mid*+1..*high*] 上**递归调用**算法即可求得仅在 *mid* 右侧的最大子数组。

(2) 求**跨越中间点的最大子数组**

因为最大子数组是“**连续**”子序列，所以**求跨越 *mid* 的最大子数组**只需基于*mid*，向左、右两侧搜索即可：



◆ **左侧**：*mid* 是**固定的终点**，从 *mid* 开始向左找 *i*，使得 $\max\limits_{1\le i\le mid} A[i..mid]$。

◆ **右侧**：*mid*+1 是**固定的起点**，从 *mid*+1 开始向右找 *j*，使得 $\max\limits_{mid<j\le high} A[mid+1..j]$。

最后，合并两侧子区间得到 $A[i..j]$，即为**跨越中间点** *mid* 的最大子数组。

以下2个过程用于求解最大子数组问题

过程1：FIND-MAX-CROSSING-SUBARRAY，求**跨越中点**的最大子数组

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
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 (max-left, max-right, left-sum + right-sum)
```

搜索从 *mid* 开始向左的半个区间，找出左侧**"和最大"**的连续子数组。

搜索从*mid*+1开始向右的半个区间，找出右边**"和最大"**的连续子数组。

返回搜索的结果

区间的开始下标

区间的终止下标

区间和

◆ 时间复杂度：$\Theta(n)$

过程2：FIND-MAXIMUM-SUBARRAY，求最大子数组问题的分治算法

```
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)
```

递归求$A[low..mid]$中的最大子数组

递归求$A[mid+1..high]$中的最大子数组

求跨越中点的最大子数组

以上三者中的大者即是区间$A[low..high]$中的最大子数组

FIND-MAXIMUM-SUBARRAY的时间分析

令 $T(n)$ 表示求解有 n 个元素的数组的最大子数组的执行时间。

（1）**当 n=1 时**，$T(1)=\Theta(1)$。否则，

（2）对 $A[low...mid]$ 和 $A[mid+1...high]$ 两个子问题递归求解，由于每个子问题的规模是 $n/2$，所以每个子问题的求解时间为 $T(n/2)$。则**递归求解两个子问题的总时间是** $2T(n/2)$。

（3）FIND-MAX-CROSSING-SUBARRAY的时间是 $\Theta(n)$。

所以 $T(n)$ 表示为：

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

思考：还有没有更快的算法？

3、最大子数组问题的时间下限

一般情况下，数组中的每个元素都可能是某个最大子数组中的元素，所以任何求最大子数组的算法都要处理到数组中的每一个元素，所以这样的算法的时间下限至少是 $\Omega(n)$。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

每个元素都要考虑到

4、最大子数组问题的线性时间算法

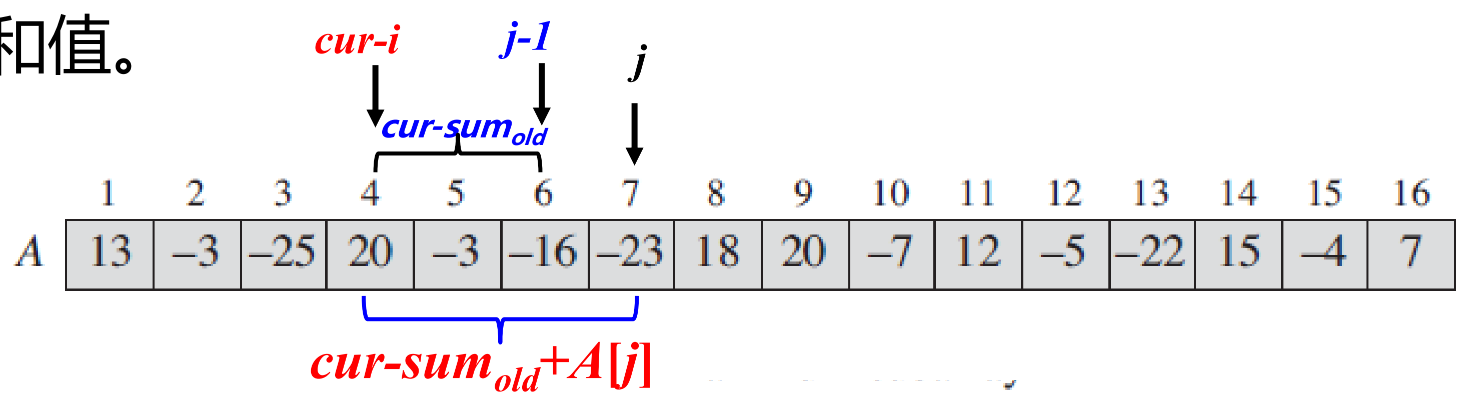
习题4.1-5给出了一个求最大子数组问题的线性时间算法

用 *max-low*、*max-high*、*max-sum* 分别记录算法当前求得的最大子数组区间的**起始下标、终止下标**和**区间和**。

算法从 $j=1$ *to* n，对每个元素做线性扫描：

当处理到 $A[j]$ 时，需判定 $A[j]$ 能否产生**更大的区间和**：

由于**子区间必须是连续**的，所以 $A[j]$ 能产生的区间和必是在之前从某个下标（记为 *cur-i* ）开始、到 j 前面的下标 j-1 处结束所获得的连续区间和（记为 *cur-sum*，初值为 -∞）的基础上加 $A[j]$ 后获得的和值。



分两种情况讨论计算到 $A[j]$ 时的**区间和**:

（1）若 $cur\text{-}sum<0$，则到 $A[j]$ 时，能获得的最大和就是 $A[j]$，所以此时仅置 $cur\text{-}i = j$，$cur\text{-}sum = A[j]$ 即可（新的开始、新的和）。

（2）否则，$cur\text{-}i$ 不变，而 $cur\text{-}sum = cur\text{-}sum + A[j]$ (无论A[j]的正负)。

证明该步的正确性

然后，比较 $max\text{-}sum$ 和 $cur\text{-}sum$:

- 若 $max\text{-}sum < cur\text{-}sum$，则更新:

  $max\text{-}low = cur\text{-}i$，$max\text{-}high = j$，$max\text{-}sum = cur\text{-}sum$;

- 否则，$max\text{-}low$、$max\text{-}high$、$max\text{-}sum$不变。

当 $j = n$ 时，算法结束，最后获得的$max\text{-}low$、$max\text{-}high$、$max\text{-}sum$即为问题的解。

上述算法的时间复杂度: $O(n)$ **课下完成习题4.1-5**

4.3 求解递归式

分治和递归是“**一对好兄弟**”：分治算法本质上是一个递归计算过程，而**分治算法的时间**通常用**递归关系式**表示和推导，如：

设原始问题的规模为 n，分解为两个规模分别为 n_1 和 n_2 的子问题。用 $T(n)$ 表示对规模为 n 的问题进行求解的时间，则递归求解规模为 n_1 和 n_2 子问题的时间可分别表示为 $T(n_1)$ 和 $T(n_2)$。

则 $T(n)$ 和 $T(n_1)$、$T(n_2)$ 的关系可表示为

$$T(n) = T(n_1) + T(n_2) + f(n)$$

其中 $f(n)$ 是计算过程中，除对两个子问题进行递归计算以外所需的时间。

- 如果 $n_1=n_2\approx n/2$，则 $T(n)$ 可表示为：

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

如归并排序：$T(n) = 2T(n/2)+cn$

- 如果 $n_1=0,\ n_2\approx n/2$（或 $n_1\approx n/2,\ n_2=0$），则 $T(n)$ 可表示为：

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

如二分查找：$T(n) = T(n/2)+1$

分治算法**原始的计算时间表达式**往往是**递归式**，如何**化简递归式**以得到形式简单的**限界函数**?

本节研究递归式的化简方法，目的是化简递归式而得到形式简单的**渐近限界函数**表达式（即用O、Ω、Θ表示的函数式）。

如 归并排序：$T(n) = 2T(n/2)+cn$ ⇨ $T(n)=O(n\log n)$

二分查找：$T(n) = T(n/2)+1$ ⇨ $T(n)=O(\log n)$

- **本节介绍三种常用的递归式求解方法**：
    - 代换法
    - 递归树法
    - 主方法

1、预处理：

为便于后续处理，对表达式的细节做一些必要、合理的假设和简化处理。

(1) 在运行时间函数 $T(n)$ 的定义中，一般假定**自变量 n 为正整数**，因为 n 通常表示数据的个数。

(2) 可以**忽略递归式的边界条件，即 n 较小时函数值的表示。**

注：虽然递归式的解会随着 $T(1)$ 值的改变而改变，但此改变对时间复杂度的影响一般不会超过**常数因子**，对**限界函数的阶**没有根本影响，所以可以忽略。

(3) 对**上取整**、**下取整**运算做合理简化。

如： $\boldsymbol{T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n)}$

通常可以忽略上、下取整函数，写成**简单形式**：

$$\boldsymbol{T(n) = 2T(n/2) + f(n)}$$

(4) 根据实际情况，**对 $f(n)$ 进行具体化。**如，

- 归并排序中，$f(n)=O(n)$，可将 $f(n)$ 展开为 cn，其中 c 是引入的常系数，从而递归式表示为 $T(n) = 2T(n/2)+cn$；
- 二分查找中，$f(n)=O(1)$，可将 $f(n)$ 直接设为**1**（或常数c），从而递归式表示为 $T(n) = T(n/2)+1$ 。

2、用代换法求解递归式

用**代换法**解递归式的基本思想是：**先猜测解的形式，然后用数学归纳法的思想验证猜测的正确性。**

此时，**用猜测的解作为归纳假设**，在推论证明时作为**较小值**代入函数证明推论的正确性，因此得名“**代换法**”。

- **用代换法解递归式的步骤：**

(1) 先猜测解的形式

(2) 再用数学归纳法证明猜测的正确性

例：用代换法确定下式的上界

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

① 猜测解的形式

- **观察**：发现该式与 $T(n) = 2T(n/2) + n$ 类似，故猜测其解为 $T(n) = O(nlogn)$ 。

② 证明猜测的正确性：

- 根据 O 的定义，若想证明 $T(n) = O(nlogn)$ ，就是设法证明

$$T(n) \le cnlogn\,,$$

也就是设法证明或说明存在一个合理的**常数** c，使得当 $n>n_0$ 时 $T(n) \le cnlogn$ 成立。

证明：

$T(n) = 2T(\lfloor n/2 \rfloor) + n$

将猜测作为归纳假设使用，从而这里假设该界对 $\lfloor n/2 \rfloor$ 情况成立，即 $T(\lfloor n/2 \rfloor) \le c(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor)$。

在推论证明阶段将 $T(\lfloor n/2 \rfloor)$ 代入递归式进行推导：

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

$$\le 2c(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor) + n$$

$$\le cn\log(n/2) + n \quad \text{去掉底函数}$$

$$= cn\log n - cn\log 2 + n \quad \text{对数运算}$$

$$= cn\log n - (c-1)n \quad \text{化简结果}$$

故，要使 $T(n) \le cn\log n$ 成立，只要 $c \ge 1$ 就可以，从而说明这样的 c **是合理存在**的。 证毕。

对边界值的讨论：

上面的过程证明了**当 n 足够大时**猜测的正确性，但对**边界值**是否成立呢？也就是 $T(n) \le cnlogn$ **的结论对于较小的 n 成立吗?**

事实上，对 $n=1$ 时，上述结论存在问题：

（1）作为边界条件，我们有理由假设 $T(1)=1$；

（2）但对$n=1$，带入表达式会有 $T(1) \le c \cdot 1 \cdot log1=0$，与$T(1)=1$ 不相符。

所以归纳证明的基础不成立了，怎么处理?

进一步讨论：从 n_0 的性质出发，只需要存在常数 n_0，使得在 $n \ge n_0$ 时结论成立即可。所以 n_0 **不一定取1**。

基于上述分析，这里不取 $n_0=1$，而取 $n_0=2$，用 $T(2)$、$T(3)$ 代替 $T(1)$ 作为归纳证明中的边界条件：

① 可以依然合理地假设 $T(1)=1$。

② 进而研究什么样的 c 可使得 $T(2)$、$T(3)$ 满足 $T(n)\le cnlogn$，

即：什么样的 c 可使 $T(2)\le 2clog2$ 且 $T(3)\le 3clog3$ 成立。

将 $T(1)=1$ 带入**原始递归式**（$T(n)=2T(n/2)+n$），则对于 $T(2)$ 有：$T(2)=2T(2/2)+2=2T(1)+2=4.$

同理，对于 $T(3)$ 有：$T(3)=5.$

故，要使 $T(2)\le 2clog2$ 和 $T(3)\le 3clog3$成立，只要 $c\ge 2$ 即可。

综上所述，取常数 $c\ge 2$，结论 $T(n)\le cnlogn$ 成立。命题得证。

◆ 如何猜测递归式的解呢？

遗憾的是，并不存在**通用的方法**来猜测递归式的正确解。

(1) 主要靠经验

- 尝试1：看有没有**形式上类似的递归式**，以此推测新递归式解的形式。
- 尝试2：先猜测一个**较宽的界**，然后再缩小不确定范围，逐步收缩到紧确的渐近界。
- **避免盲目推测**

如：对 $T(n)=2T(\lfloor n/2\rfloor)+n$，猜测为 $T(n)=O(n)$ 对吗？

似乎有 $T(n)\le 2(c\lfloor n/2\rfloor)+n\le cn+n=O(n)$ 成立，但这个猜测是 **错误的**。因为 $cn+n \nleq cn$，无法证出其一般形式 $T(n)\le cn$ 成立。

（2）必要的时候可以做一些技术处理

思路1：去掉低阶项（详见书上的例子）

思路2：变量代换

对陌生的递归式做一些简单的**代数变换**，使之变成熟悉的形式，然后再进行求解。

例，设有递归式 $T(n) \le 2T(\lfloor\sqrt{n}\rfloor) + logn$

分析：**原始形态比较复杂**

① **做代数变换**：令 $m = logn$，则 $n=2^m$，$\sqrt{n} = 2^{m/2}$；

② **忽略下取整**，直接使用 $\sqrt{n}$ 代替 $\lfloor\sqrt{n}\rfloor$，从而得

$$T(2^m) \le 2T\left(2^{m/2}\right) + m$$

再令 $S(m)=T(2^m)$，进而得出以下形式的递归式：

$$S(m)\le 2S(m/2)+m$$

$T(2^m)\le 2T(2^{m/2})+m$

$S(m)$ 就是形式上"**熟悉**"的递归式

猜测 $S(m)$ 的上界并完成证明： $S(m)=O(m\log m)$。

再将 $T(n)=S(m)$、$m=\log n$ 带回 $S(m)$ 的表达式，有：

$$\begin{aligned}T(n)&=T(2^m)\\&=S(m)=O(m\log m)\\&=O(\log n\log\log n)\end{aligned}$$

这里，$m=\log n$ ■

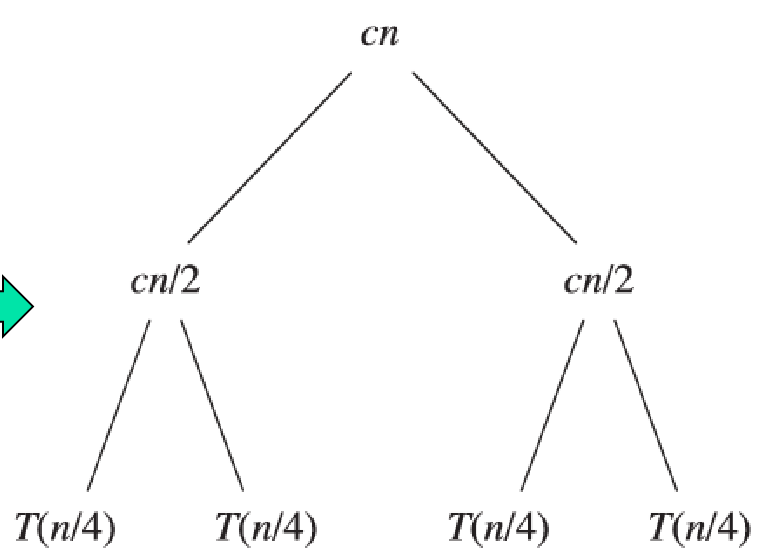
3、用递归树法求解递归式

- 用一棵树描述递归算法的执行过程，称为**递归树**。

在递归树中，结点表示各级递归调用对应的问题或子问题，根结点代表顶层调用时的原始问题。**从上至下表示问题的分解过程**（产生子问题），而**从下至上表示递归返回的过程**（合并子问题解）。

如：

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



一棵递归树

基于递归树的时间分析

① **结点代价**：每个结点有求解该结点对应子问题的**代价**。

这里**结点代价**仅指除递归以外的其它代价（递归的代价在后续各层里体现）。

② **层代价**：一层内所有结点的**结点代价**之和。

③ **树的总代价**：整棵树各层代价之和，或树中所有结点的结点代价之和。

利用递归树进行分析，就是**利用树的性质**获取该**树总代价**的表示，并将其作为对递归式解的一种**猜测**，然后用代换法或其它方法来验证猜测的正确性。

例：已知递归式 $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$，求其上界。

(1) 准备性工作：（对细节做一些简化和假设）

① **去掉底函数的表示**。

② **假设 n 是4的幂**，即$n=4^k$，$k=\log_4 n$。

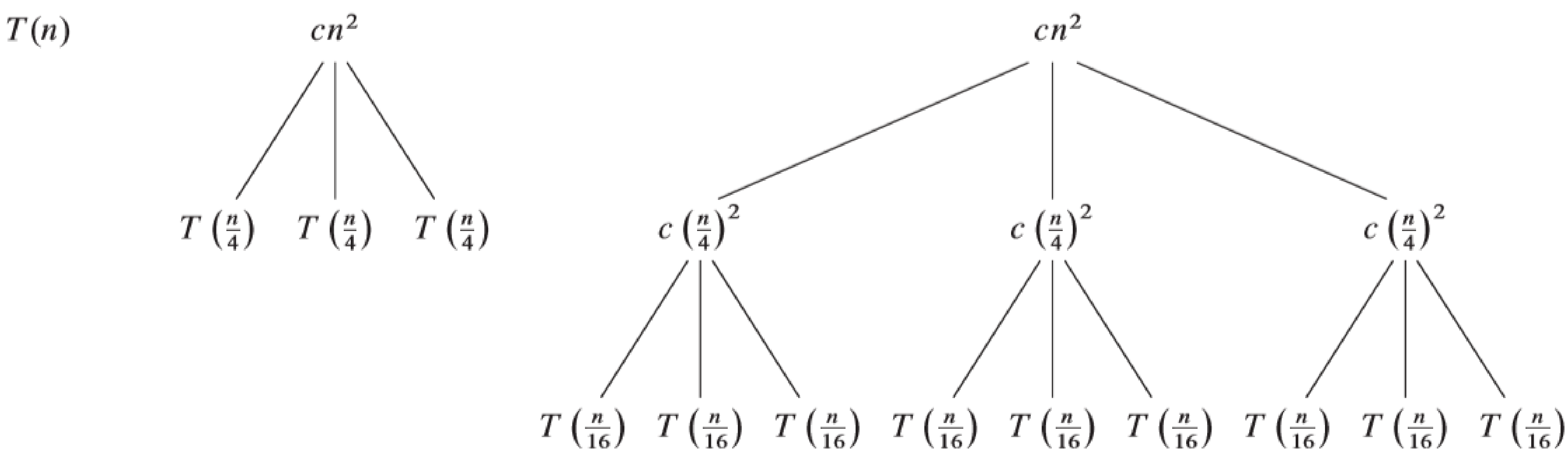
——在证明 $n=4^k$ 的情况结论成立后，再推广到 $n\neq 4^k$ 的情形，就可以把结论从特殊推广到一般情况了。

③ **展开$\Theta(n^2)$**

——$\Theta(n^2)$ 代表了算法中**非递归部分的计算时间**，但 Θ 的是**抽象符号**，不能直接运算。所以化简时通常**根据定义**，引入常系数 c，$c>0$，将其转变成 cn^2 的形式（对 O 和 Ω 也有类似的处理）。

最终得到的是以下形式的递归式：$T(n) = 3T(n/4) + cn^2$

(2) 用递归树描述 $T(n) = 3T(n/4) + cn^2$ 的演化过程:



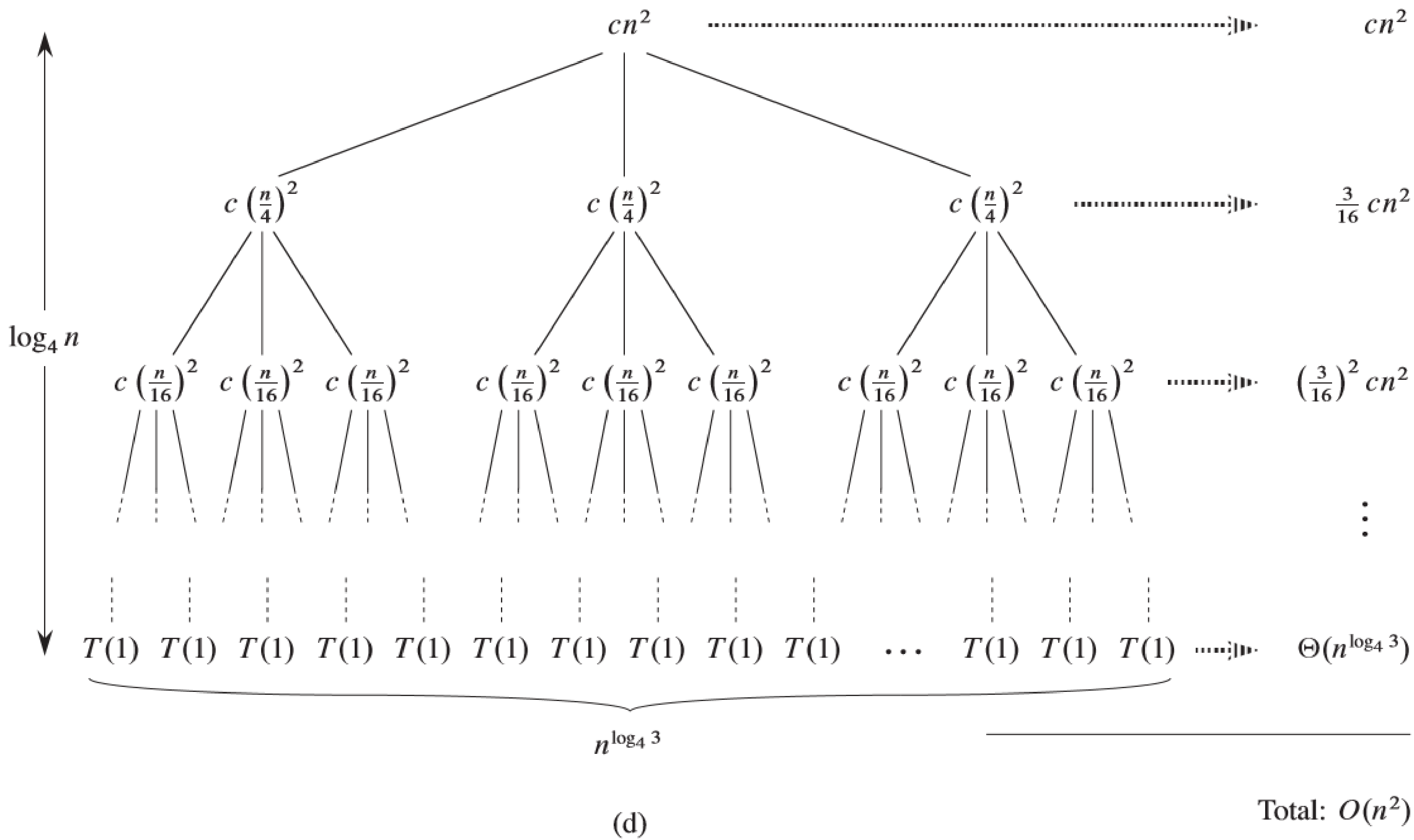
(a) (b) (c)

(a) 是对原始问题计算时间 $T(n)$ 的表示。

(b) 是第一次分解子问题并递归调用的情况，cn^2 是根结点的结点代价，代表求解初始问题时除递归计算以外的其它代价。$T(n/4)$ 是分解出来的规模为 $n/4$ 的子问题的递归代价（总代价 $T(n) = 3T(n/4) + cn^2$）。

(c) 是第一级子问题分解后第二级递归调用的情况。$c(n/4)^2$ 是第一级三棵子树计算中除递归以外的其它代价。

继续扩展下去，直到**最底层**，得到如下形式的递归树：



(d)表示完全扩展后的的递归树。

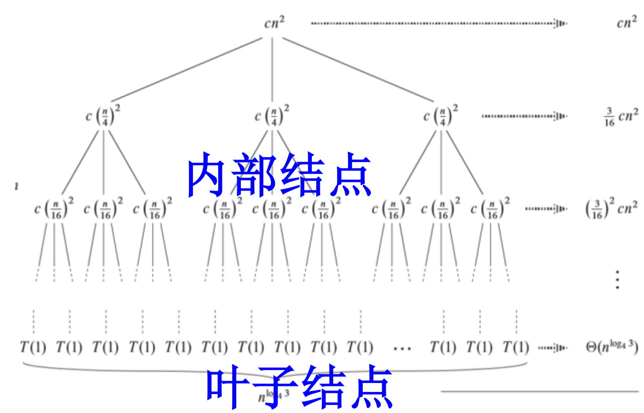
递归树的高度为 $\log_4 n$，共有 $\log_4 n+1$ 层。

子问题的规模按 **1/4** 的比例不断减小，在递归树中，深度为 i 的结点对应的子问题的大小为 $n/4^i$， $0 \le i \le \log_4 n$。

当 $n/4^i = 1$ 时，子问题规模为1，到达边界值。所以该递归树有如下性质：

- 结点分布在 $0 \sim \log_4 n$ 层；
- 该树共有 $\log_4 n+1$ 层；
- 每一层上的结点数是其上一层结点数的 **3 倍**（根结点除外）；
- 深度为 i 的层中结点数为 3^i。

代价的计算



① **内部结点（非叶子结点）**：

- 内结点位于$0 \sim \log_4 n\text{-}1$层
- 深度为 i 的内结点的代价为 $c(n/4^i)^2$; 非递归部分
- i 层内共有 3^i 个内结点，所以 i 层中所有结点的代价之和为

$$3^i c(n/4^i)^2 = (3/16)^i cn^2$$

② **叶子结点：** 位于$\log_4 n$层，共有$3^{\log_4 n} = n^{\log_4 3}$个，

- 每个叶子结点的代价为 $T(1)$;
- 所以叶子结点总代价为 $n^{\log_4 3}T(1) = \Theta(n^{\log_4 3})$

③ **树的总代价**：等于各层代价之和，则有

$$T(n) = \underbrace{cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \ldots\ldots + (\frac{3}{16})^{\log_4 n-1} cn^2}_{\text{内结点总代价}} + \underbrace{\Theta(n^{\log_4 3})}_{\text{叶子层}}$$

$$= \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

利用**等比数列**化简上式。

- 对于实数$x \neq 1$，和式 $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$ 是一个**几何级数**(等比数列)，其值为 $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$.
- 当和式是**无穷数列且 $|x|<1$** 时，该级数是一个**无穷递减几何级数**，此时有

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$T(n)$ 中，cn^2 项的系数构成一个**递减的几何级数**：

$$T(n) = \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

将 $T(n)$ 扩展到无穷

$$< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

至此，获得 $T(n)$ 解的一个猜测：$T(n) = O(n^2)$。**成立吗?**

下面用代换法证明猜测的正确：

将 $T(n) \le dn^2$ 作为归纳假设，带入推论证明过程。这里 d **是待确定的常数**。具体有：

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$
$$\le 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + cn^2$$
$$\le 3d\lfloor n/4 \rfloor^2 + cn^2$$
$$\le 3d(n/4)^2 + cn^2$$
$$= \frac{3}{16}dn^2 + cn^2$$

c是引入的另一确定存在的常量

显然，要使得 $T(n) \le dn^2$ 成立，只要 $d \ge (16/13)c$ 即可，因为 c 存在，所以这样的 d 是合理存在的。所以 $T(n) = O(n^2)$ 得证。

（边界条件的讨论略。另，**$\Theta(n^2)$是T(n)的一个紧确界**，为什么？见教材P52）

例 求表达式 $T(n) = T(n/3) + T(2n/3) + O(n)$ 的上界

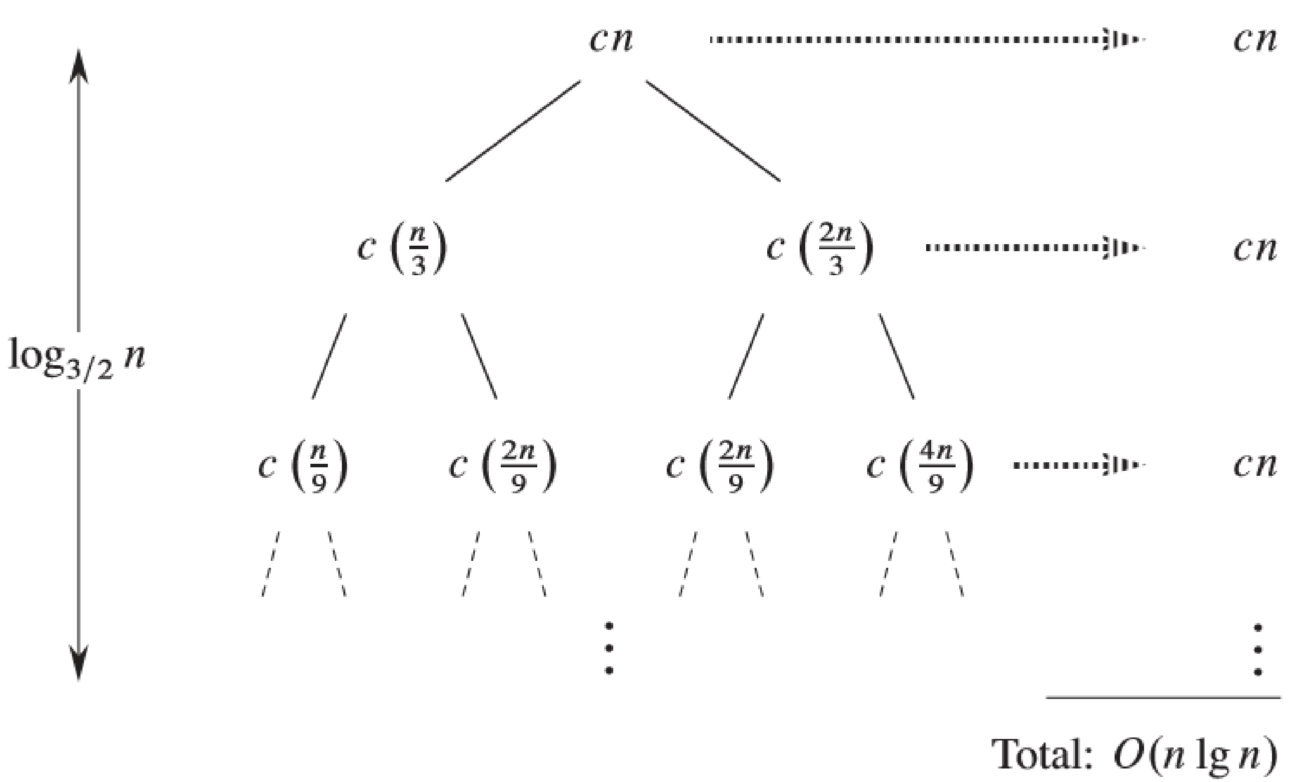
(这里，表达式中直接省略了下取整和上取整函数)

① **预处理**：**引入常数c**，展开$O(n)$，得：

$$T(n) \le T(n/3) + T(2n/3) + cn$$

层代价

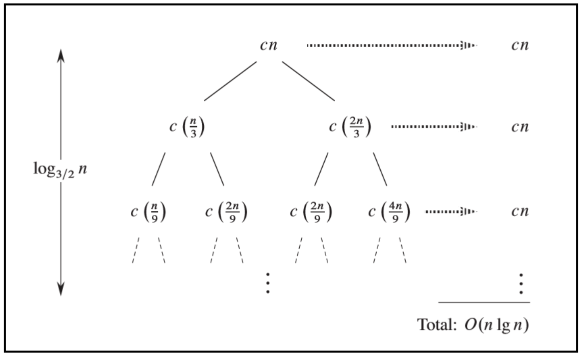
② **建立递归树**：



注意所划分的子问题的大小是不相等的。

分析：

- **该树并不是一个完全的二叉树。**
  - 从根往下，越来越多的内节点**在左侧消失**（1/3分叉上），因此每层的代价并不总是 cn，而是 $\leq cn$ 的某个值。
  - 但这里可以视 cn **为每层代价的上界。**



- **树的深度：**
  - 在上述形态中，**最长的路径是最右侧路径**，由

$$n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \ldots \rightarrow 1$$

组成。

  - 当 $k=\log_{3/2} n$ 时，$(3/2)^k/n=1$，所以**树的深度为** $\log_{3/2} n$ 。

③ **递归式解的猜测**：

每层的代价是 $O(n)$，总共有 $O(\log_{3/2}n)$ 层，所以可猜测总代价（上界）为**层数乘以每层的代价**。

由于层代价 $\leq cn$、树的高度 $\leq \log_{3/2}n$，所以：

$$T(n) \leq cn\log_{3/2}n = O(n\log n)$$

④ **猜测的证明**：

即证明会存在常数 $d > 0$，使得 $T(n) \leq dn\log n$ 成立。

具体如下。

证明 $T(n) \le dn\log n$， d 是**待确定**的合适的正常数：

$$T(n) \le T(n/3) + T(2n/3) + cn$$

c是引入的正常量

$$\le d(n/3)\log(n/3) + d(2n/3)\log(2n/3) + cn$$

$$= (d(n/3)\log n - d(n/3)\log 3) + (d(2n/3)\log n - d(2n/3)\log 3/2)) + cn$$

$$= dn\log n - d((n/3)\log 3 + (2n/3)\log(3/2)) + cn$$

$$= dn\log n - d((n/3)\log 3 + (2n/3)\log 3 - (2n/3)\log 2) + cn$$

$$= \underline{dn\log n - dn(\log 3 - 2/3) + cn} \le dn\log n$$ 成立吗?

上式的成立条件是：只要 $d \ge c/(\log 3 - (2/3))$ 即可，这样的d是**合理存在**的！

$\therefore$ **猜测正确**。递归式解得证。

4、用主方法求解递归式

用**主方法**求解递归式是指用**“主定理”的结论**直接给出一个递归式的解，方法简单，但有限制。

- 主方法只适合求解具有如下形式的递归式：

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

这里，a、b 是常数，且 $a \geq 1$，$b>1$；

$f(n)$ 是一个渐近正的函数；

这里做了细节简化，省略了下取整、上取整。

只有上述形式的递归式才能用主方法求解。

(1) 主定理

定理4.1 主定理：设 $a\geq 1$ 和 $b>1$ 为常数，设 $f(n)$是一个渐近正的函数，$T(n)$是定义在非负整数上的递归式：

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

其中 n/b 指 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$

则 $T(n)$ 可有如下的**渐近界**：

① 若对于某常数 $\varepsilon > 0$，有 $f(n) = O(n^{\log_b a-\varepsilon})$，则

$$T(n) = \Theta(n^{\log_b a});$$

② 若$f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a}\log n)$；

③ 若对某常数 $\varepsilon > 0$，有 $f(n) = \Omega(n^{\log_b a+\varepsilon})$，且对常数$c<1$和所有足够大的 n，有 $af(n/b) \leq cf(n)$，则 $T(n) = \Theta(f(n))$。

◆ 理解主定理：

① 对于 $T(n) = aT(n/b) + f(n)$ 形式的递归式，$T(n)$ 的解与 $f(n)$ 和 $n^{\log_b a}$ 有密切关系：**$T(n)$ 取其中（数量级）较大的一个**。

- **第一种情况**，函数 $n^{\log_b a}$ 比较大，所以 $T(n) = \Theta(n^{\log_b a})$；
- **第三种情况**，函数 $f(n)$ 比较大，所以 $T(n) = \Theta(f(n))$；
- **第二种情况**，两个函数一样大，此时乘上对数因子 $\log n$，有 $T(n) = \Theta(n^{\log_b a} \log n)$；

② 在第一种情况中，$f(n)$ 要 **“多项式”** 地小于 $n^{\log_b a}$ 。

即：存在常量 $\varepsilon > 0$，$f(n)$ 不仅渐近小于 $n^{\log_b a}$，而且两者相差一个n^{ε}因子。

③ 在第三种情况中，$f(n)$ 不仅要大于 $n^{\log_b a}$，而且要“**多项式**”地大于 $n^{\log_b a}$，并要满足一个“**规则性**”条件：存在 $c < 1$，使得：

$$af(n/b) \le cf(n)。$$

④ 若递归式中的 $f(n)$ 与 $n^{\log_b a}$ 的关系不满足上述性质，特别是：

- $f(n)$ 小于等于 $n^{\log_b a}$，但**不是多项式地小于**。
- $f(n)$ 大于等于 $n^{\log_b a}$，但**不是多项式地大于**。

则**不能用主方法求解该递归式**。

（2）主方法的使用

首先**分析递归式满足主定理三种情况的哪一种**，然后根据**定理的结论**给出相应的解即可 (无需证明，保证正确）。

例4.6 解递归式 $T(n) = 9T(n/3) + n$

解：这里 a=9，b=3，故有

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$ 。

又因为 $f(n)=n$，故有

$$f(n) = n = O(n^{\log_3 9-\varepsilon})$$，这里可取 ε=1、0.5等。

所以对应主定理的**第一种情况**。

于是有：$T(n) = \Theta(n^2)$

例4.7 解递归式 $T(n)=T(2n/3)+1$

解：这里 a=1，b=3/2，故有

$$n^{\log_b a}=n^{\log_{3/2}1}=n^0=1。$$

又因为 $f(n)$=1，故有

$$f(n)=\Theta(n^{\log_b a})=\Theta(1)。$$

所以对应主定理的**第二种情况**。

于是有：$T(n)=\Theta(\log n)$。

例4.8 解递归式 $T(n) = 3T(n/4) + n\log n$

解：这里 $a=3$，$b=4$，故有

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

又因为 $f(n)=nlogn$ ，故有

$$f(n) = \Omega(n^{\log_4 3+\varepsilon})$$，这里可取 $\varepsilon \approx 0.2$。

$0.793 + 0.2 = 0.993 < 1$

同时，对足够大的 n，有

$$af(n/b) = 3(n/4)\log(n/4) \le (3/4)n\log n = cf(n)$$

其中，取 $c = 3/4$。

所以对应主定理的**第三种情况**。

于是有：$T(n) = \Theta(n\log n)$。

例4.9 递归式 $T(n)=2T(n/2)+n\log n$ **不能用主方法求解。**

分析：这里 $a=2$，$b=2$，于是有 $n^{\log_b a}=n^{\log_2 2}=O(n)$；

且，$f(n)=n\log n$ **渐进大于**$n^{\log_b a}=O(n)$，

但**第三种情况成立吗？事实上不成立！** **不是多项式大于**

原因： 对于任意正常数 ε，

$$f(n)/n^{\log_b a}=(n\log n)/n=\log n<n^{\varepsilon}$$

此处引用了性质：
$n^x(\log n)^y<n^{x+\varepsilon}$

不满足 $f(n)=\Omega(n^{\log_b a+\varepsilon})$

注：若要 $f(n)=\Omega(n^{\log_b a+\varepsilon})$，需有 $f(n)/n^{\log_b a}>n^{\varepsilon}$。

因此该递归式**落在情况二和情况三之间**，条件不成立，不能用主定理求解。

（3）证明主定理

为什么主定理是正确的?

主定理证明：（略，见教材P55）

注：**使用主定理时不需要证明其正确性。**

还有没有其它方法化简递归式?

5、直接化简

直接展开递归式，找出各项系数的构造规律（如等差数列、等比数列等），然后得出化简后的最终形式。

如：$\boldsymbol{T(n) = 2T(n/2) + 2}$

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

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

$$\ldots$$

$$= 2^{k\text{-}1}T(2) + \sum_{1\le i\le k-1} 2^i$$

$$= 2^{k\text{-}1}+2^k\text{-}2$$

$$=\boldsymbol{3n/2\text{-}2 = O(n)}$$

例：化简递归式 $\boldsymbol{T(n)=2T(n/2)+n\log n}$。

解：

$$
\begin{aligned}
\boldsymbol{T(n)} &= \boldsymbol{2T(n/2)+n\log n}\\
&= 2(\boldsymbol{2T(n/4)+(n/2)\log(n/2)})+n\log n\\
&= 2^2T(n/2^2)+\boldsymbol{n\log n-n}+n\log n\\
&= 2^2T(n/2^2)+\boldsymbol{2n\log n-n}\\
&= 2^2(2T(n/2^3)+(n/4)\log(n/4))+2n\log n-n\\
&= 2^3T(n/2^3)+\boldsymbol{n\log n-2n}+2n\log n-n\\
&= 2^3T(n/2^3)+\boldsymbol{3n\log n-2n-n}\\
&=\ldots
\end{aligned}
$$

$\boldsymbol{k=logn}$

$$
\begin{aligned}
&= \underline{2^kT(n/2^k)}+kn\log n-n\sum_{i=1}^{k-1} i\\
&= n+kn\log n-n(k-1)k/2\\
&= n+n\log^2 n-(n/2)\log^2 n+n\log n/2\\
&= \boldsymbol{O(n\log^2 n)}
\end{aligned}
$$

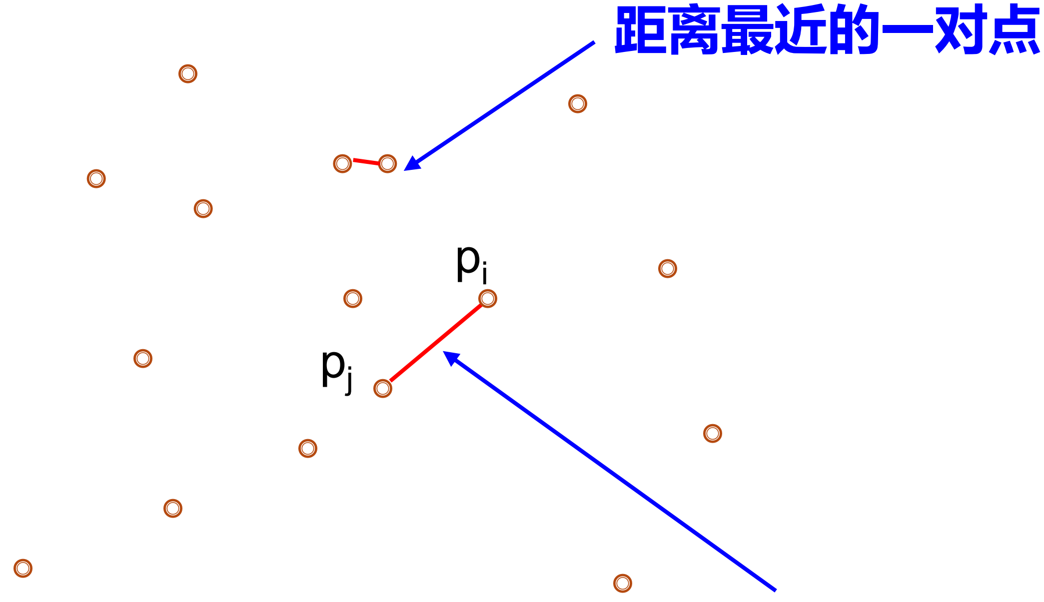
33.4 最近点对问题

1. 问题描述

已知平面上分布着**点集*P***中的 n 个点 $p_1, p_2, \dots p_n$，点 i 的坐标记为 (x_i, y_i)，$1 \le i \le n$。两点之间的距离取其**欧式距离**，记为：

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

请找出一对距离最近的点。



$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

平面上分布着点集P，找出其中相距最近的一对结点。

允许两个点位于同一个位置，此时两点之间的距离为0。

方法1：暴力搜索

- 对**每对点**都计算距离并比较距离的大小，找出其中的最小者。
- **该方法的时间复杂度为 $O(n^2)$：**
  - 计算所有结点对之间的距离： $O(n^2)$

    —— 因为共有 $C_2^n = n(n-1)/2$ 对点要计算间距。
  - 找最小距离：$O(n^2)$

    —— 因为需要做 $n(n\text{-}1)/2\text{-}1$ 次比较。

所以，总的时间复杂度：$O(n^2)$ 。

有没有更好的办法?

2、求解最近点对问题的分治算法

我们想利用**分治法**设计一个 $O(n\log n)$ 时间复杂度的算法求解最近点对问题。

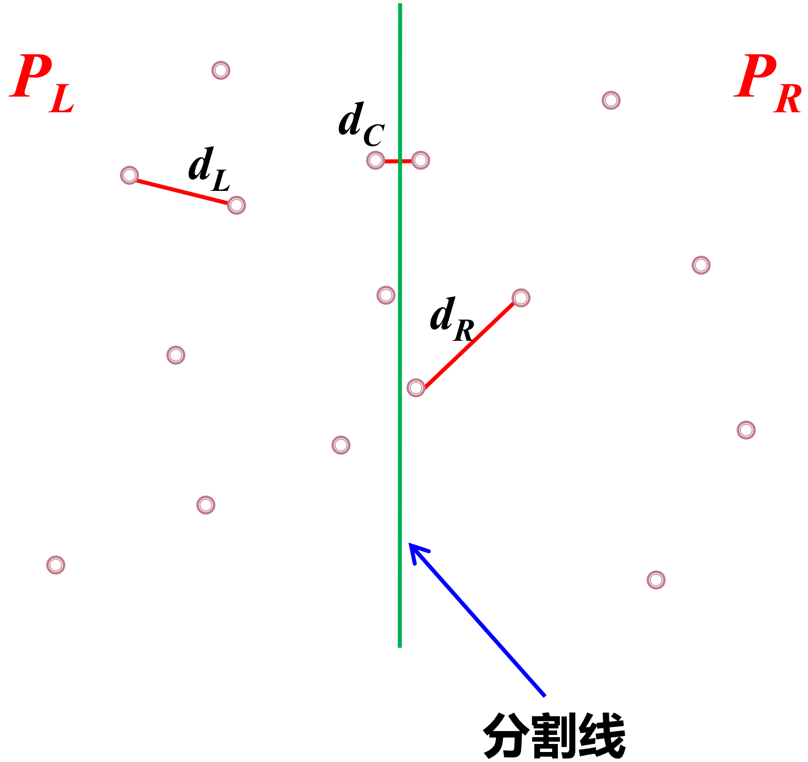
（1）设计策略：分治

① 首先将所有的点按照 x 坐标的大小排序

这个排序过程需要 $O(n\log n)$ 的时间，但**不会增加算法的整体时间复杂度。**

② 划分

以 x **坐标的中间值**作一条分割线（**中垂线**），将平面 P 分成左、右两半部分，分别记为 P_L 和 P_R。如下所示。



平面 P 中最近的一对点可能出现的位置：

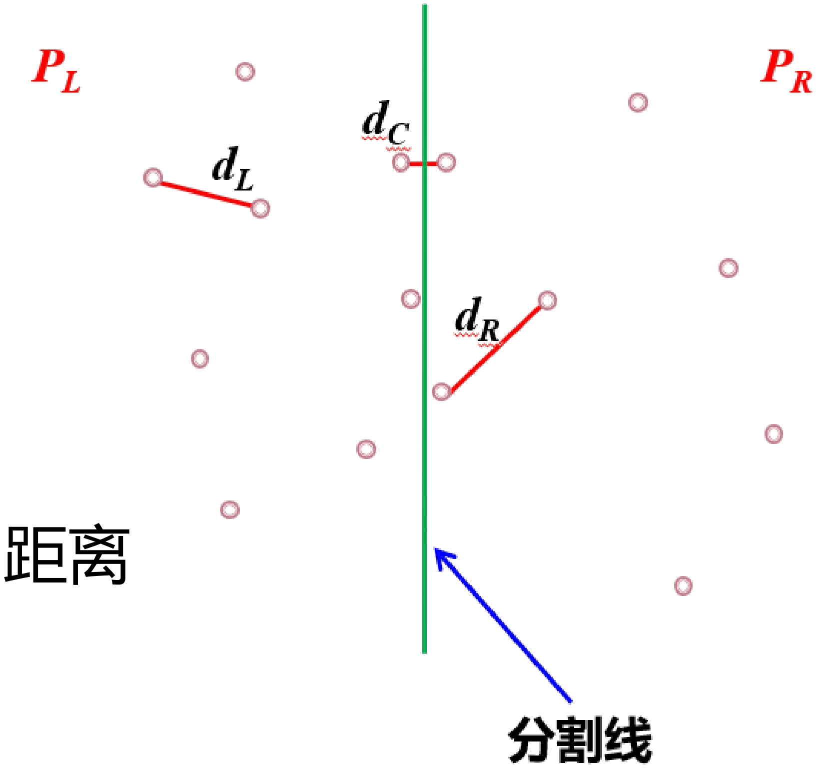
- 或者**在 P_L 中**：两个点都在 P_L 中；
- 或者**在 P_R 中**：两个点都在 P_R 中；
- 或者**跨越中垂线** (分割线)：一个点在 P_L 中，另一个端点在 P_R 中。

记:

d_L: P_L中最近点对的距离

d_R: P_R中最近点对的距离

d_C: **跨越分割线**情况下的最近点对的距离



设计一个算法，求出 d_L、 d_R、 d_C，然后**比较三者的大小**，其中**最小者对应的点对**就是平面 P 中的最近点对。

③ 计算 d_L、 d_R、 d_C 及其对应的最近点对

设**算法** A 是求平面 P 中最近点对的算法。

(1) 如何求 P_L 和 P_R 中的最近点对?

—— 在 P_L 和 P_R 中**递归调用算法** A 即可求出仅在 P_L 或 P_R 中的最近点对及其对应的 d_L、 d_R。

(2) 如何求跨越分割线的最近点对及其 d_C？且**至多只能花** $O(n)$ **的时间**，这样才能使总时间为：

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

从而使算法的总时间控制在 $O(n\log n)$ 以内。

基于上述思路，算法的流程大致如下：

$A(P, d, pt)$

(1) 对 P 中的点按 x **坐标**排序，然后做分割线，将 P 分为 P_L 和 P_R 两部分；

(2) $A(P_L, d_L, pt_L)$； // 在 P_L 中递归求解，求出 d_L 及其对应的点对 pt_L；

(3) $A(P_R, d_R, pt_R)$； //在 P_R 中递归求解，求出 d_R 及其对应的点对 pt_R；

(4) 求跨越分割线的 d_C 及其对应的点对 pt_C;

(5) $d = min(d_L, d_R, d_C)$； $pt = d$ **对应的点对**； //pt_L、pt_R、pt_C三者之一

(6) $return\ d, pt$

该如何实现？上述流程中，**(2)、(3)是递归过程**，“认为”是可以完成。**关键是(4)**：求跨越分割线时的 d_C及其对应的点对？

如何求跨越分割线时的 d_C 及其对应的点对?

暴力搜索：对 P_L 中的每个点求其到 P_R 中所有点的距离（或反之）。

但**时间复杂度**是：$(n/2)*(n/2) = O(n^2)$，**超时!**

进一步分析:

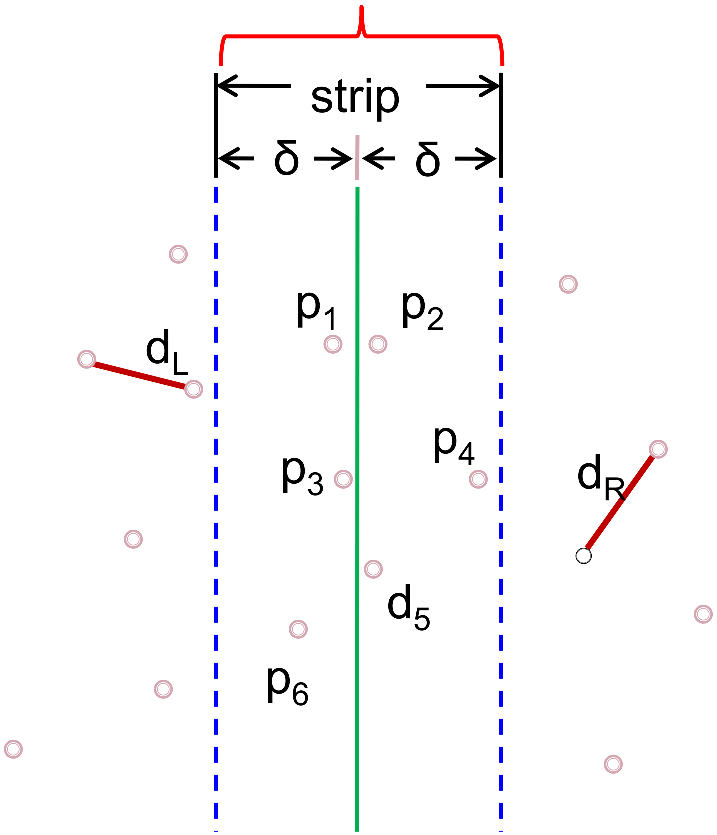
在上述流程中，**d_L 和 d_R 在计算 d_C 之前已经求出**（程序流程是先做对 P_L 和 P_R 的递归，递归过程返回后才求本级的 d_C）。

令 $\delta=min(d_L, d_R)$，则计算 d_C 时，任何时刻都应有 $d_C < \delta$，即搜索过程中，任何距离大于 δ 的结点对都不用考虑，而**只考虑相距小于 δ 的结点**。

这样的点对只会分布在分割线两侧的 δ 距离以内。如下。

◆ d_C对应的点仅分布在分割线两侧的 δ 距离以内的区域：

把这个区域叫做**带** (*strip*)



d_C只会出现在 *strip* 内

(1) **带外的点不用考虑。**

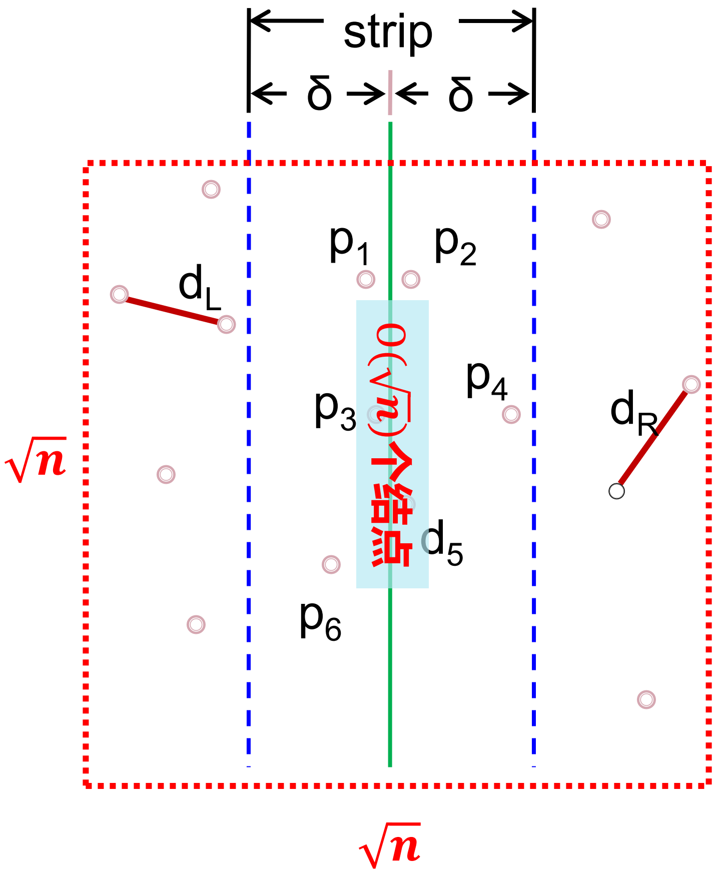
因为它们到分割线的距离大于 δ，不可能构成**跨越分割线时更近的点对**，所以在计算跨越分割线的最近点对时带外的点都**不用考虑**。

(2) **只在 *strip* 区域内搜索 d_C**

这样就大大减少了搜索中需要考虑的点的个数。

计算 d_C

如果点的分布是均匀、稀疏的，则**可预计**位于该带中的点是"均匀"且"稀疏"的；而 δ 是个 **"很小"** 的值，所以**分割线两侧 δ 范围内分布的点**平均都只有 $O(\sqrt{n})$ 个。



朴素的计算过程：

$$
\begin{aligned}
&\textit{for } i=1 \textit{ to numPointsInStrip do} \\
&\quad \textit{for } j=i+1 \textit{ to numPointsInStrip do} \\
&\qquad \textit{if } \ \mathbf{dist}\,(p_i, p_j) < \delta \\
&\qquad\quad \delta = \mathbf{dist}\,(p_i, p_j);
\end{aligned}
$$

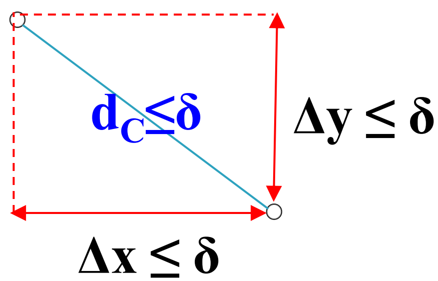
δ 不断被修正

这里，$\textbf{numPointsInStrip} = O(\sqrt{n})$，因此可以在 $\sqrt{n} \times \sqrt{n} = O(n)$ 时间内计算出 δ.

但如果**点的分布不均匀、不稀疏**，则分割线两侧**带内的点**的个数就无法保证在$O(\sqrt{n})$内，即 $numPointsInStrip > O(\sqrt{n})$（甚至$>> O(\sqrt{n})$），那么**上述过程超时！**不能保证在 $O(n)$ 时间内完成。

再进一步分析：

- 上述讨论的是分割线两侧**“水平”方向**的距离不超过 δ；
- **垂直方向呢?** 事实上，两点垂直方向的距离也不应大于δ：



即：一个点仅需与 “$\Delta x \leq \delta$ **且**$\Delta y \leq \delta$”范围内的点计算距离，超出这个范围的点没必要计算。

改进的计算过程：

for i=1 **to** *numPointsInStrip* **do**

for j=i+1 **to** *numPointsInStrip* **do**

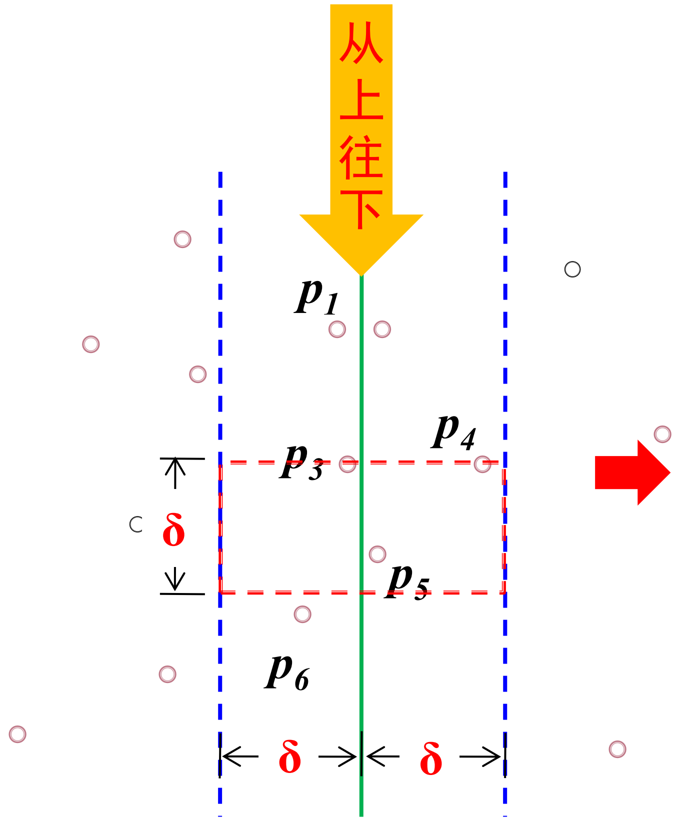
if $|y_{p_i} - y_{p_j}| > \delta$ **break**; $\Delta y < \delta$

else if $dist\,(p_i, p_j) < \delta$

$\delta = dist\,(p_i, p_j)$;

为此，同时**增加对 y 坐标的排序**，使得在计算**带内的每个点**到其它点的距离的时候，**只往下搜寻对端结点即可。**

如图所示：



如：对于 p_3，只考虑 p_4 和 p_5，不用**向上**考虑 p_1；也不用考虑 p_6，因为 **p_6 在 δ 范围之外**。

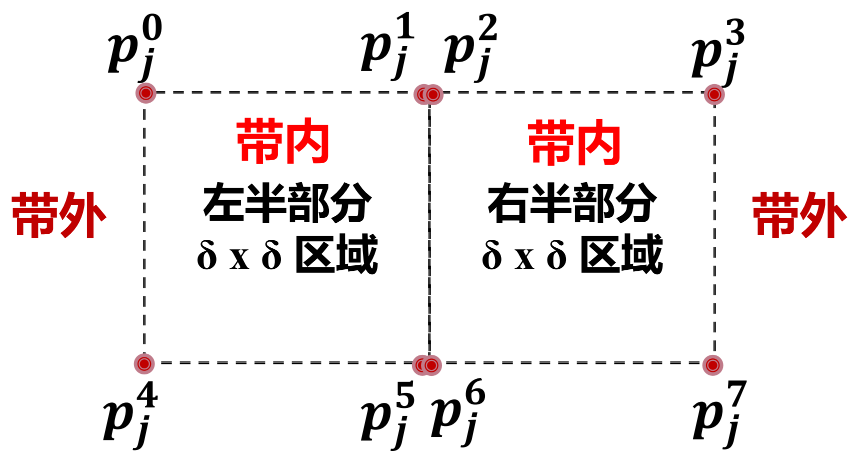
显著的改进：对每个 p_i **大大减少**了需考察的对端结点 p_j **的数量**。

到底会考察几个点呢?

——最坏情况只需要考虑**7个**。

- 一般情况下，基于点p_i计算点对距离时，最坏情况下只需要考虑**7个** p_j。

如下所示：在 $\delta \times 2\delta$ 的搜索区域内，最多有8个点，**其中一个是 p_i 自己**，**另外7个**（如果存在的话）就是要与 p_i 计算距离的点——最多7个。



最坏情况的示意图

为什么？见后

现在先假设上述分析是正确的（即对任何 p_i，最多和另外7个点计算距离——**确实是正确的**）。

这样对每个 p_i 最多就计算**7个距离**，所以对 p_i 自身而言完成一次这样的计算时间复杂度就是 $O(1)$ 。

再考虑带内的所有点，**哪怕点的分布不均匀**：最坏情况下，哪怕有 $O(n)$ **个**点集中分割线两侧，每个点都会成为 p_i 而计算到另外7个点的距离，但由于每个点仅需7次计算($O(1)$**的时间**)，所以计算 d_C 总的时间也仅是 $O(n) \times O(1) = O(n)$。

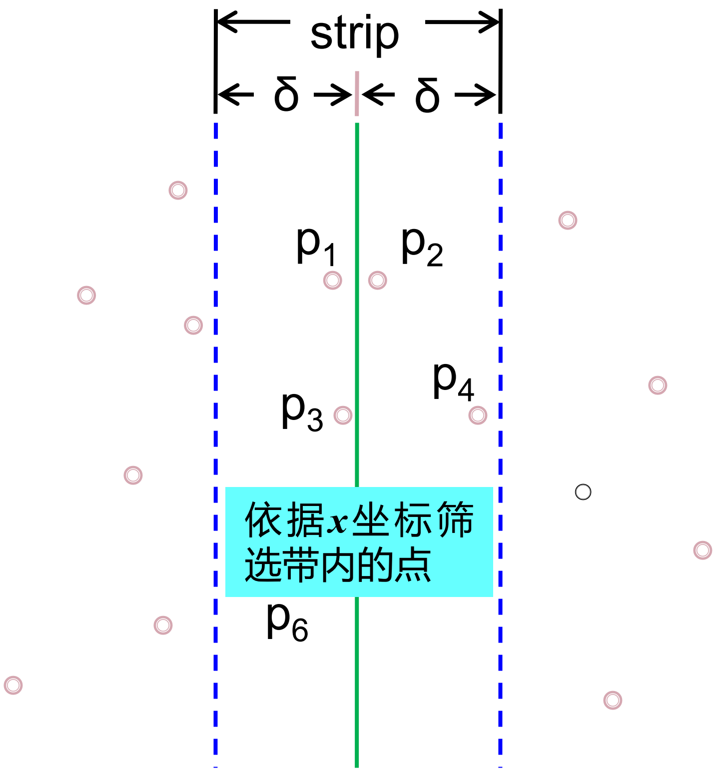
于是可得：$T(n) = 2T(n/2) + O(n)$

这样，就可使总的时间达到 $O(n\log n)$。

最近点对的求解过程是两个 $T(n/2)$ 时间的递归调用加上 $O(n)$ 时间的跨分割线计算组成。

◆ 但还存在一个的问题：

把**平面划分为左右两个半平面**是依据 x **坐标**进行的，找出**带内的点**也要按 x **坐标**进行（因为**带是垂直的**）。



如果只是按照 x 坐标筛选，得到的仅是**按 x 坐标有序**的点集，y **坐标上是无序的**。而上面的设计要求带内的点按 y **坐标**有序。怎么办？

朴素的方法：

筛选出带内的点后，现做一次 y 坐标的排序。

相应的计算流程可描述为：

collect the points in the strip; $O(n)$

sort the points in the strip by *y*-coordinate; $O(n\log n)$

for $i=1$ **to** *numPointsInStrip* **do**

for $j=i+1$ **to** *numPointsInStrip* **do**

if $|y_{p_i} - y_{p_j}| > \delta$ **break**;

else if $dist(p_i,p_j)<\delta$

$\delta = dist(p_i,p_j)$;

$O(n)$

这样，在计算 d_C 之前要附加 **$O(n\log n)$ 的排序时间**，所以总的时间就变为：

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) + \boldsymbol{O(n\log n)}$$

collect the points in the strip; $O(n)$

sort the points in the strip by *y*-coordinate; $O(n\log n)$

for i=1 **to** *numPointsInStrip* **do**

for j=i+1 **to** *numPointsInStrip* **do**

if $|y_{p_i} - y_{p_j}| > \delta$ **break**;

else if $dist(p_i,p_j)<\delta$

$\delta = dist(p_i,p_j)$;

$O(n)$

整个算法的时间复杂度就变成 $O(n\log^2 n)$。 **怎么办?**

解决方案：**预排序**，具体如下：

(1) 对原始问题，先预处理两个表：

- ***P* 表**：对所有点**按** x **坐标排序**后得到的表；
- ***Q* 表**：对所有点**按** y **坐标排序**后得到的表。

—— 这两个表都可以在 $O(n\log n)$ 时间内得到。

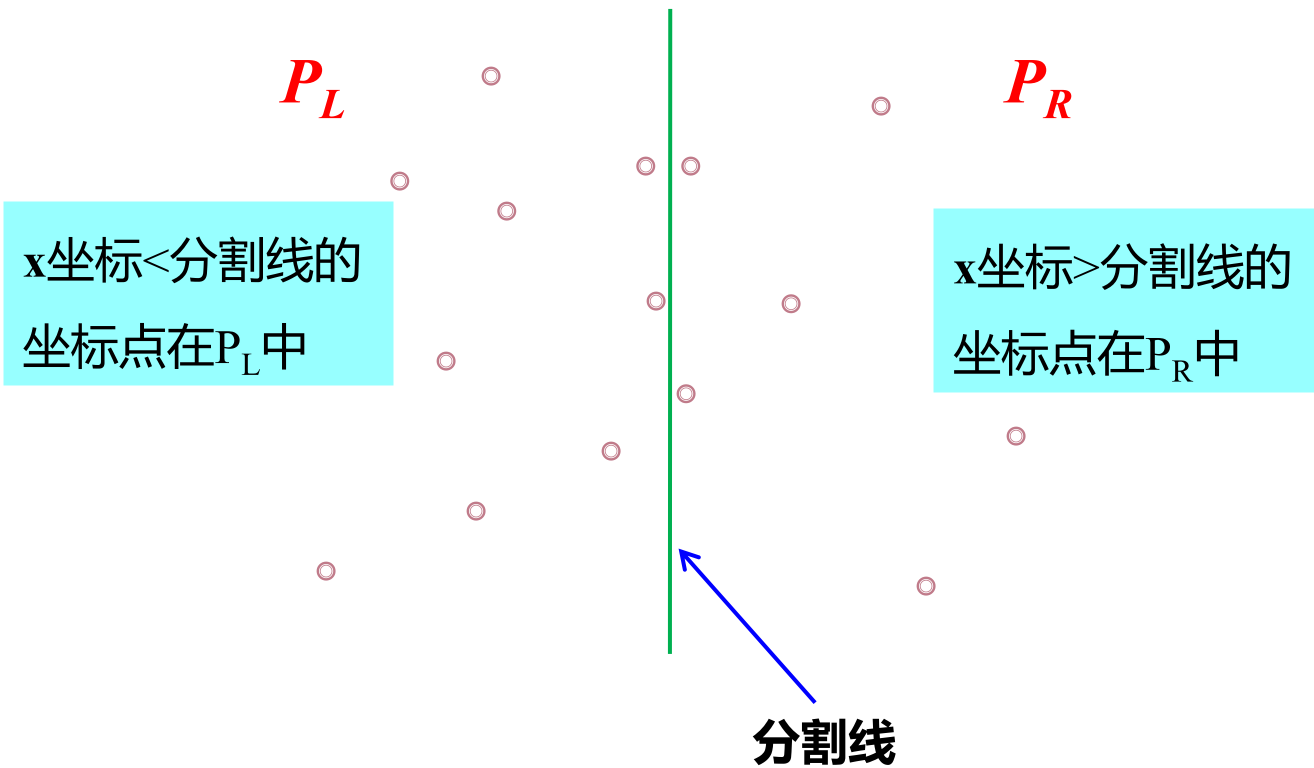
然后进入计算过程。

(2) 对子平面进行递归处理前，先构造**四个子表：**

- P_L 和 Q_L：从 P 表中筛选仅在左半部分的点组成P_L，从 Q 表中筛选仅在左半部分的点组成 Q_L。P_L 和Q_L 就是分别按照 x 和 y 坐标排好序且仅在左半部分的点组成的子表。
- P_R 和 Q_R：同理收集仅在右半部分的点组成这两个子表。

◆ 如何筛选点构成 P_L 和 Q_L、P_R 和 Q_R?

分割线确定后，把**每个点的x坐标和分割线的位置坐标（一条关于x的垂线）比较一下**，就可将 P 中的点筛选进 P_L 和 P_R、将 Q 中的点筛选进 Q_L 和 Q_R 中。**这些操作可在 $O(n)$ 时间内完成。**



(3) 之后将P_L和 Q_L、P_R和 Q_R分别传递给左半部分和右半部分的递归过程进行递归处理。

(4) 递归返回后，“**带**”就确定了，即**“带”左、右两侧的边界坐标**就计算出来了。再以“带”的边界坐标**筛选 Q 表**中的点得到另一个子表 Q_C，Q_C **中就仅包含 Q 中在当前带中的点** ——**这一操作同样可在$O(n)$时间内完成。**

而且 Q_C 中的点已按 y 坐标排好序，然后就简单地从上至下依次计算“跨分割线”的最近结点对距离即可。

(5) d_C计算完成后，再比较 d_L、d_R、d_C，执行过程结束。

改进后的算法的流程描述：

① **预处理**：将原始点集分别按 x 和 y 坐标排序，得到 P 和 Q。

② ClosestPointPair (P, Q, d, pt) //P和Q分别是已按 x 坐标和 y 坐标排好序的点集

(1) 基于 P 计算分割线; $O(1)$

(2) 将 P 筛选进 P_L 和 P_R；将 Q 筛选进 Q_L 和 Q_R； $O(n)$

(3) ClosestPointPair (P_L, Q_L, d_L, pt_L)； // 在 P_L 中递归求解; $T(n/2)$

(4) ClosestPointPair (P_R, Q_R, d_R, pt_R)； //在 P_R 中递归求解; $T(n/2)$

(5) **计算带**，并**从 Q 中筛选出带内的点，得到 Q_C；** $O(n)$

(6) 基于 Q_C 计算“跨越分割线”的最近点对pt_C及 d_C； $O(n)$

(7) $d = min(d_L, d_R, d_C)$; $O(1)$

$pt = d$ **对应的点对**；

(8) *return*

综上所述，整个算法的计算时间为

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= O(n \log n)
\end{aligned}
$$

注：上述过程中，

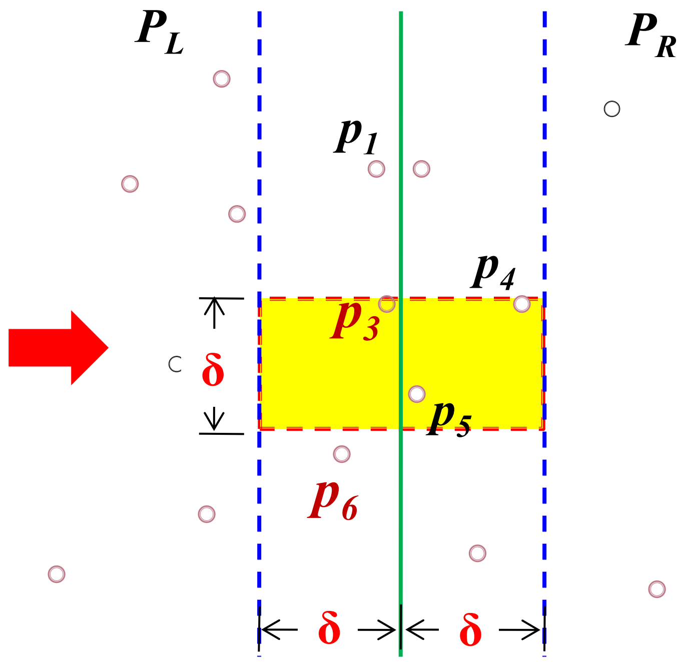
- 可以**将 P 表作为全局表使用**，从而在递归过程中直接使用 P 表而不需要单独构造子表 P_L **和** P_R，以减少生成 P_L 和 P_R 表的时间和空间。
- 但从 Q 构造 Q_L、Q_R、Q_C 仍是必需的，**尤其是** Q_C。思考为什么？

再回到“7个点”的问题上。

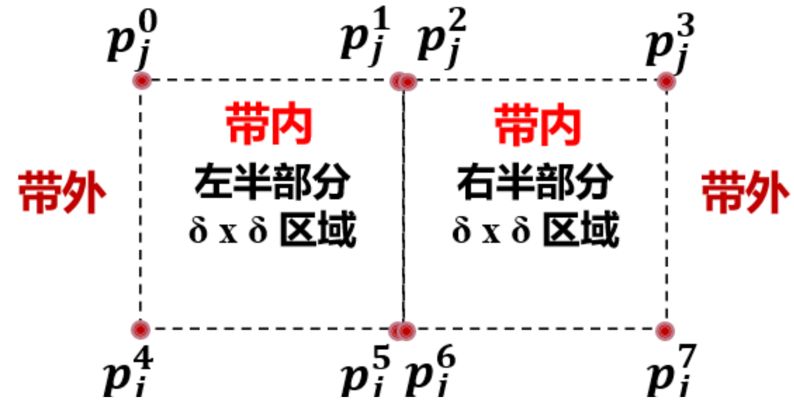
在上述设计的计算 d_C 过程中，基于某个点 p_i 计算点对距离时，最坏情况下只与另外**7个** p_j 计算*dist*。**为什么?**

分析:

因为是在 "**带内**" 且**从上往下计算**，所以无论 p_i 是带内的哪个点，计算时最多只考虑分割线**左、右两侧** $\Delta x \le \delta$ 及 p_i **下方** $\Delta y \le \delta$ 的“$\delta\times 2\delta$”区域内的点。即如图所示的黄色矩形区域。

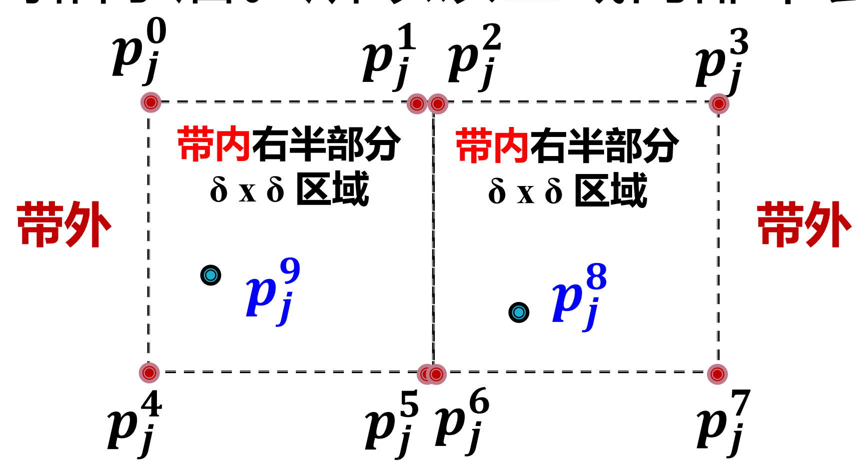


而这样的“**δ×2δ**”矩形区域内，分布“最远”的最多就是这样的8个点：

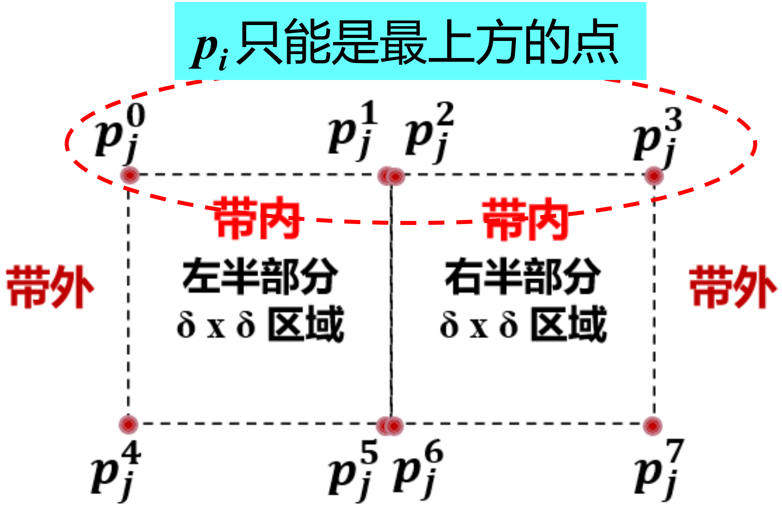


区域外的点不用考虑，区域内就没有其它的点吗?

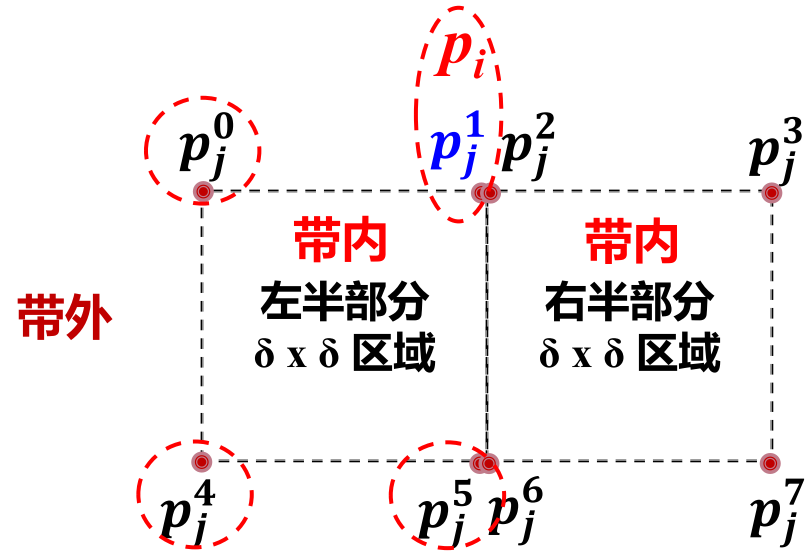
反证：如下所示，如果区域内部还有其它点，如 $\boldsymbol{p_j^8}$、$\boldsymbol{p_j^9}$，则 p_j^8、p_j^9 到周围p_j^0~ p_j^7中一些点的距离就会小于带宽$\boldsymbol{\delta}$，这与$\boldsymbol{\delta}$至少是递归返回后获得的最小距离相矛盾。所以该区域内部不会有像 p_j^8、p_j^9 这样的点存在。



另外，根据**从上至下**的计算规则，p_i 不仅是这8个点中之一，而且还**只能是最上方的点**。



再一个问题：不是计算**跨分割线**情况下的点对距离嘛，为什么不是最多与另一侧的4个点计算距离，而还要考虑同侧的点?

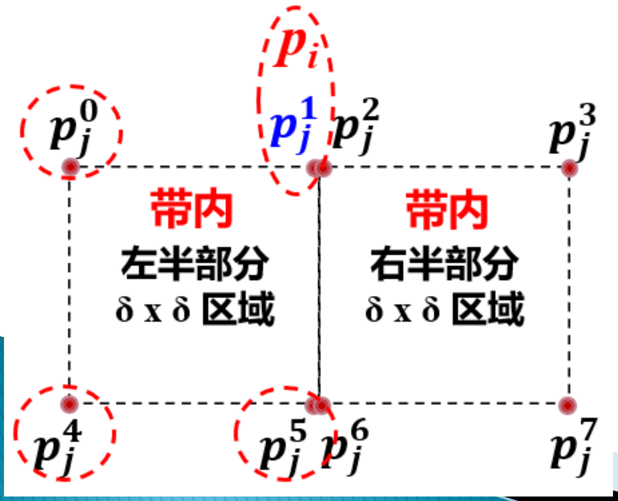


如图，不失一般性，设p_j^1就是 p_i，则它除了会和可能存在的p_j^2、p_j^3、p_j^6、p_j^7 计算，还会和可能存在的 p_j^0、p_j^4、p_j^5 计算。这是**由计算过程决定**的。

计算过程的第(5)步：

(5) **计算带**，并**从 Q 中筛选出带内的点，得到 Q_C；**

在该步从 Q 中筛选结点集 Q_C 时，虽然用 x **坐标**判断一个点是否在带内，但 Q 中的点是按 y **坐标排序**的，所以 Q_C 中的点也仅是 y 方向有序，x 方向是无序的。而且“从上往下”计算时，也只考虑了 $\Delta y \le \delta$ 的点，并没有对点处于分割线哪一侧进行判断。所以即使像 p_j^0、p_j^4、p_j^5，虽然它们和 p_i 处于同一侧，但因为它们和 p_i 的 $\Delta y \le \delta$，所以一样会被计算到——**最终是“7个点”**，虽然会多一点工作量，但不会影响算法的本质。



只判断$\Delta y \le \delta$

改进的计算过程：

for i=1 to numPointsInStrip do

for j=i+1 to numPointsInStrip do

if $|y_{p_i} - y_{p_j}| > \delta$ *break;* ⟵ $\Delta y < \delta$

else if $dist(p_i, p_j) < \delta$

$\delta = dist(p_i, p_j)$;

4.2 Strassen矩阵乘法

已知两个 n **阶方阵**，记为 $A=(a_{ij})_{nxn}$， $B=(b_{ij})_{nxn}$

1、矩阵运算的数学规则：

（1） **矩阵加法**

$C=A+B=(c_{ij})_{nxn}$， 其中，$c_{ij}=a_{ij}+b_{ij}$, $i, j=1, 2, \ldots, n$

时间复杂度：$\Theta(n^2)$

（2） **矩阵乘法**

$C=A\times B=(c_{ij})_{nxn}$， 其中，$c_{ij}=\sum_{1\le k\le n} a_{ik}b_{kj}$， $i,j=1,2,\ldots,n$

时间复杂度：$\Theta(n^3)$。

2、两个 $n \times n$ 矩阵乘的基本过程

SQUARE-MATRIX-MULTIPLY(A, B)　　//朴素的矩阵乘法

```
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} .$$

朴素矩阵乘法的计算时间是 $\Theta(n^3)$.

能否用少于 $\Theta(n^3)$ 的时间完成矩阵乘计算？

1969年，德国数学家 **Strassen**： $O(n^{2.81})$

3、Strassen 矩阵乘法：基于**分治策略**的矩阵乘算法

(1) 基本情况：两个 2×2 矩阵相乘

① 朴素的矩阵乘计算过程：

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\begin{aligned} C = AB &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ &= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \\ &= \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \end{aligned}$$

直接相乘共需要8次元素乘和4次元素加

② Strassen 矩阵乘的计算方法

先计算七个量（记为*P*、*Q*、*R*、*S*、*T*、*U*、*V*），然后再通过这七个量的**加、减组合**来生成 *C* 矩阵中的四个元素：

$P=(a_{11}+a_{22})(b_{11}+b_{22})$

$Q=(a_{21}+a_{22})\ b_{11}$

$R=a_{11}\ (b_{12}-b_{22})$

$S=a_{22}\ (b_{21}-b_{11})$

$T=(a_{11}+a_{12})b_{22}$

$U=(a_{11}-a_{21})\ (b_{11}+b_{12})$

$V=(a_{12}-a_{22})\ (b_{21}+b_{22})$

$$A=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix},\qquad B=\begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\end{bmatrix}$$

$$C=AB=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}\begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\end{bmatrix}$$
$$=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22}\\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22}\end{bmatrix}$$
$$=\begin{bmatrix}c_{11} & c_{12}\\ c_{21} & c_{22}\end{bmatrix}$$

计算 *C* 矩阵元素：

$c_{11}=\ P+S-T+V \equiv a_{11}b_{11}+a_{12}b_{21}$

$c_{12}=\ R+T \equiv a_{11}b_{12}+a_{12}b_{22}$

$c_{21}=\ Q+S \equiv a_{21}b_{11}+a_{22}b_{21}$

$c_{22}=\ P+R-Q-U \equiv a_{21}b_{12}+a_{22}b_{22}$

验证：

$$c_{11}=P+S-T+V$$
$$=(a_{11}+a_{22})(b_{11}+b_{22})$$
$$+a_{22}(b_{21}-b_{11})-(a_{11}+a_{12})b_{22}$$
$$+(a_{12}-a_{22})(b_{21}+b_{22})$$
$$=a_{11}b_{11}+a_{12}b_{21}$$

两个 2×2 矩阵相乘Strassen计算方法的分析：

$P=(a_{11}+a_{22})(b_{11}+b_{22})$
$Q=(a_{21}+a_{22})\ b_{11}$
$R=a_{11}\ (b_{12}-b_{22})$
$S=a_{22}\ (b_{21}-b_{11})$
$T=(a_{11}+a_{12})b_{22}$
$U=(a_{11}-a_{21})\ (b_{11}+b_{12})$
$V=(a_{12}-a_{22})\ (b_{21}+b_{22})$

这里，a_{ij}、b_{ij} 是基本元素，为求*P*、*Q*、*R*、*S*、*T*、*U*、*V* 共需**10次元素加(减)** (括号之内)、**7次元素乘** (括号之间)。

计算 *C* 矩阵元素：

c_{11} = **P+S-T+V**
c_{12} = **R+T**
c_{21} = **Q+S**
c_{22} = **P+R-Q-U**

这里，*P*、*Q*、*R*、*S*、*T*、*U*、*V*是**标量**（数值），为求 $c_{11}\sim c_{22}$，需**8次元素加（减）**。

- **总的计算量：7次**乘、**18次**加（减）。
- 与朴素的计算过程比较，增加了14次加（减），减少了1次乘。

——— 通过**减少**乘法、适当**增加**加法，从“**总体上**”减少矩阵乘的运算时间。

（2）一般情况下的两个$n\times n$矩阵相乘

不失一般性，设 $n = 2^k$ （$k\geq1$，若 $n\neq2^k$ ，可通过补0的方式使矩阵变成阶是**2的幂**的方阵）

① 若 k=1（即 A 和 B 都是2×2的“最小”矩阵），则按照上面的基本情况直接计算即可。

② 否则，首先将 A 和 B 各分成 4 个 $(n/2)\times(n/2)$ 的子矩阵：

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

然后采用**矩阵分块相乘**的规则进行计算。

方法1：朴素的分治思想 —— 简单的矩阵分块相乘

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

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

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

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

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

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

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

共有：

- **8次** $(n/2)\text{x}(n/2)$ 子矩阵乘
- **4次** $(n/2)\text{x}(n/2)$ 子矩阵加

注：**任意两个子矩阵块的乘可以沿用同样的规则**：如果子矩阵的阶大于2，则将子矩阵分成更小的子矩阵，直到每个子矩阵只含一个元素（或为2×2的基本矩阵）为止。从而可以构造出一个递归计算过程。

简单矩阵分块相乘的时间分析：

令 $T(n)$ 表示两个 $n \times n$ 矩阵相乘的计算时间。

则首次分块时需要：

1) 8次 $(n/2)\times(n/2)$ 子矩阵乘，所需时间为 $8T(n/2)$

2) 4次 $(n/2)\times(n/2)$ 子矩阵加，所需时间为 dn^2 ，d 为常系数。

所以，**简单矩阵分块相乘**的时间为：

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + dn^2 & n > 2 \end{cases}$$

其中，b，d 是常数

化简得：$T(n) = O(n^3)$ 与矩阵"直接相乘"相同

- **Strassen矩阵乘的一般方法：**与 2×2 基本矩阵乘类似，**先计算七个量**（同样记为*P*、*Q*、*R*、*S*、*T*、*U*、*V*），然后也是通过**组合这七个量**来计算最后的结果矩阵，注：这里的 *P*~*V* 七个量均是$(n/2)\times(n/2)$**的子矩阵。**

$$A=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\qquad B=\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}\qquad C=AB=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}$$

令

$$P=(A_{11}+A_{22})(B_{11}+B_{22})$$

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

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

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

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

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

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

计算结果矩阵：

$$C_{11}=P+S-T+V$$

$$C_{12}=R+T$$

$$C_{21}=Q+S$$

$$C_{22}=P+R-Q-U$$

一般情况下Strassen矩阵乘的分析

$$P = (A_{11}+A_{22})(B_{11}+B_{22})$$
$$Q = (A_{21}+A_{22})B_{11}$$
$$R = A_{11}(B_{12}-B_{22})$$
$$S = A_{12}(B_{21}-B_{11})$$
$$T = (A_{11}+A_{12})B_{22}$$
$$U = (A_{11}-A_{21})(B_{11}+B_{12})$$
$$V = (A_{12}-A_{22})(B_{21}+B_{22})$$

为求P、Q、R、S、T、U、V 共需**10次** $(n/2)\times(n/2)$ **子矩阵加（减）**（括号之内）、**7次** $(n/2)\times(n/2)$**子矩阵乘**(括号之间)。

计算结果矩阵：

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q - U$$

为求 $C_{11}\sim C_{22}$，需 **8次** $(n/2)\times(n/2)$ **矩阵加（减）**。

- 总共需要 7次 $(n/2)\times(n/2)$ 矩阵乘， 18次 $(n/2)\times(n/2)$ 矩阵加。

注：Strassen矩阵乘也是一个递归求解过程， 任意两个子矩阵的块乘沿用同样的规则进行。

Strassen矩阵乘法的时间：

令 $T(n)$ 表示两个 $n=2^k$ 阶方阵的Strassen矩阵乘所需的计算时间，则有：

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

其中， a 和 b 是常数

化简：

$$\begin{aligned} T(n) &= an^2(1+7/4+(7/4)^2+\ldots+(7/4)^{k-1}) + 7^k T(1) \\ &\le cn^2(7/4)^{logn}+7^{logn} \\ &= cn^2 n^{log(7/4)} + n^{log7} \\ &= cn^{log4+log7-log4} + n^{log7} \\ &= (c+1)n^{log7} \\ &= O(n^{log7}) \approx O(n^{2.81}) \end{aligned}$$

这里，$k = logn$

也可以用主方法直接化简递归式

Strassen矩阵乘的C++实现代码如下：

```cpp
#include <iostream>

using namespace std;

const int N = 6;    //Define the size of the Matrix

template<typename T>
void Strassen(int n, T A[][N], T B[][N], T C[][N]);

template<typename T>
void input(int n, T p[][N]);

template<typename T>
void output(int n, T C[][N]);

int main() {
   //Define three Matrices
   int A[N][N],B[N][N],C[N][N];

   //对A和B矩阵赋值，随便赋值都可以，测试用
   for(int i=0; i<N; i++) {
     for(int j=0; j<N; j++) {
        A[i][j] = i * j;
        B[i][j] = i * j;
     }
   }

   //调用Strassen方法实现C=A*B
   Strassen(N, A, B, C);

   //输出矩阵C中值
   output(N, C);

   system("pause");
   return 0;
}

/**The Input Function of Matrix*/
template<typename T>
void input(int n, T p[][N]) {
    for(int i=0; i<n; i++) {
      cout<<"Please Input Line "<<i+1<<endl;
      for(int j=0; j<n; j++) {
        cin>>p[i][j];
      }
    }
}

/**The Output Fanction of Matrix*/
template<typename T>
void output(int n, T C[][N]) {
    cout<<"The Output Matrix is :"<<endl;
    for(int i=0; i<n; i++) {
      for(int j=0; j<n; j++) {
        cout<<C[i][j]<<" "<<endl;
      }
    }
}

/**Matrix Multiplication as the normal algorithm*/
template<typename T>
void Matrix_Multiply(T A[][N], T B[][N], T C[][N]) { //Calculating A*B->C
    for(int i=0; i<2; i++) {
      for(int j=0; j<2; j++) {
        C[i][j] = 0;
        for(int t=0; t<2; t++) {
          C[i][j] = C[i][j] + A[i][t]*B[t][j];
        }
      }
    }
}

/**Matrix Addition*/
template <typename T>
void Matrix_Add(int n, T X[][N], T Y[][N], T Z[][N]) {
    for(int i=0; i<n; i++) {
      for(int j=0; j<n; j++) {
        Z[i][j] = X[i][j] + Y[i][j];
      }
    }
}

/**Matrix Subtraction*/
template <typename T>
void Matrix_Sub(int n, T X[][N], T Y[][N], T Z[][N]) {
    for(int i=0; i<n; i++) {
      for(int j=0; j<n; j++) {
        Z[i][j] = X[i][j] - Y[i][j];
      }
    }
}

/**
* 参数n指定矩阵A,B,C的阶数，因为随着递归调用Strassen函数
* 矩阵A,B,C的阶数是递减的N只是预留足够空间而已
*/
template <typename T>
void Strassen(int n, T A[][N], T B[][N], T C[][N]) {
    T A11[N][N], A12[N][N], A21[N][N], A22[N][N];
    T B11[N][N], B12[N][N], B21[N][N], B22[N][N];
    T C11[N][N], C12[N][N], C21[N][N], C22[N][N];
    T M1[N][N], M2[N][N], M3[N][N], M4[N][N], M5[N][N], M6[N][N], M7[N][N];
    T AA[N][N], BB[N][N];

    if(n == 2) {  //2-order
      Matrix_Multiply(A, B, C);
    } else {
      //将矩阵A和B分成阶数相同的四个子矩阵，即分治思想。
      for(int i=0; i<n/2; i++) {
        for(int j=0; j<n/2; j++) {
          A11[i][j] = A[i][j];
          A12[i][j] = A[i][j+n/2];
          A21[i][j] = A[i+n/2][j];
          A22[i][j] = A[i+n/2][j+n/2];

          B11[i][j] = B[i][j];
          B12[i][j] = B[i][j+n/2];
          B21[i][j] = B[i+n/2][j];
          B22[i][j] = B[i+n/2][j+n/2];
        }
      }

      //Calculate M1 = (A0 + A3) × (B0 + B3)
      Matrix_Add(n/2, A11, A22, AA);
      Matrix_Add(n/2, B11, B22, BB);
      Strassen(n/2, AA, BB, M1);

      //Calculate M2 = (A2 + A3) × B0
      Matrix_Add(n/2, A21, A22, AA);
      Strassen(n/2, AA, B11, M2);

      //Calculate M3 = A0 × (B1 - B3)
      Matrix_Sub(n/2, B12, B22, BB);
      Strassen(n/2, A11, BB, M3);

      //Calculate M4 = A3 × (B2 - B0)
      Matrix_Sub(n/2, B21, B11, BB);
      Strassen(n/2, A22, BB, M4);

      //Calculate M5 = (A0 + A1) × B3
      Matrix_Add(n/2, A11, A12, AA);
      Strassen(n/2, AA, B22, M5);

      //Calculate M6 = (A2 - A0) × (B0 + B1)
      Matrix_Sub(n/2, A21, A11, AA);
      Matrix_Add(n/2, B11, B12, BB);
      Strassen(n/2, AA, BB, M6);

      //Calculate M7 = (A1 - A3) × (B2 + B3)
      Matrix_Sub(n/2, A12, A22, AA);
      Matrix_Add(n/2, B21, B22, BB);
      Strassen(n/2, AA, BB, M7);

      //Calculate C0 = M1 + M4 - M5 + M7
      Matrix_Add(n/2, M1, M4, AA);
      Matrix_Sub(n/2, M7, M5, BB);
      Matrix_Add(n/2, AA, BB, C11);

      //Calculate C1 = M3 + M5
      Matrix_Add(n/2, M3, M5, C12);

      //Calculate C2 = M2 + M4
      Matrix_Add(n/2, M2, M4, C21);

      //Calculate C3 = M1 - M2 + M3 + M6
      Matrix_Sub(n/2, M1, M2, AA);
      Matrix_Add(n/2, M3, M6, BB);
      Matrix_Add(n/2, AA, BB, C22);

      //Set the result to C[][N]
      for(int i=0; i<n/2; i++) {
        for(int j=0; j<n/2; j++) {
          C[i][j] = C11[i][j];
          C[i][j+n/2] = C12[i][j];
          C[i+n/2][j] = C21[i][j];
          C[i+n/2][j+n/2] = C22[i][j];
        }
      }
    }
}
```

- **Strassen**算法发表于1969年，为快速求解矩阵乘提供了新思路。
- 但从实用的角度，**Strassen**算法并不是解决矩阵乘的最好选择：

  （1）隐藏在**Strassen**算法运行时间 $\Theta(n^{\log 7})$ 中的常数因子比直接过程的 $\Theta(n^3)$ 的常数因子大。

  （2）对于稀疏矩阵，有更快的专用算法可用。

  （3）**Strassen**算法的数值稳定性不如直接过程，其计算过程中引起的误差积累比直接过程大。

  （4）递归过程生成的子矩阵会消耗更多的存储空间。
- 不断地在改进。见P63的分析讨论。目前已知的 $n \times n$ 矩阵乘的最优时间是 $O(n^{2.376})$。

分治法给予的一些启示：

（1）如果能够**二分处理**，可使算法的时间复杂度“**降级**”到**对数级**或“**准**”**对数级**，而且这样的算法的时间复杂度函数里一般都有 $\log n$因子，如 $O(\log n)$、$O(n\log n)$。

（2）**最近点对问题**：**代价和收益共存**。花小代价做一些预处理可以显著提高整体收益。

（3）**Strassen矩阵乘法**：利用子问题的“**局部性**”和“**金字塔**”型求解，可以利用小问题的局部解来改进和优化后续大问题的求解，从而获得整体加速，实现更高效的求解。

本章作业

(1) 2.4： 逆序数对问题

(2) 4.3-2：证明递归式 $T(n) = T(\lceil n/2 \rceil) + 1$ 的解是 $O(\lg n)$。

(3) 4.3-9：利用改变变量的方法求解递归式 $T(n) = 3T(\sqrt{n}) + logn$ 。

得到的解应是紧确的。

(4) 4.4-6：对递归式 $T(n) = T(n/3) + T(2n/3) + cn$ 利用递归树证其

解是 $\Omega(n\,log\,n)$ ，其中 c 是一个常数。

(5) 4.5-1：用主方法来给出下列递归式的紧确渐近界:

b) $T(n) = 2T(n/4) + n^{1/2}$

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

(6) 4.5-4：主方法能否应用于递归式 $T(n)=4T(n/2)+n^2logn$?

为什么？试用其它方法推导其渐近上界。

(7) 4.1-5：求解最大子数组问题的线性时间算法