

算法设计与分析

Computer Algorithm Design & Analysis



吕志鹏

zhipeng.lv@hust.edu.cn

群名称：算法设计与分析

群 号：271069522

群名称：算法设计与分析2020

群 号：271069522



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，将原始问题分解为几个规模较小、但类似于原始问题的子问题，然后递归地求解这些子问题，最后合并子问题的解以得到原始问题的解。

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，形式与原问题一样的子问题；
- 2) **解决 (Conquer)**：若子问题规模较小、可直接求解时则直接解；否则“递归”地求解各个子问题，即继续将较大子问题分解为更小的子问题，然后重复上述计算过程。
- 3) **合并 (Combine)**：将子问题的解合并成原问题的解。

分治算法的实例：归并排序

对一个含有 n 个元素无序序列进行排序：

■ 归并排序的基本思路：

分解：将含有 n 个元素的待排序分解成两个各具 $n/2$ 个元素的子序列。

解决：递归地对两个子序列进行排序，以得到两个排序子序列。

合并：合并两个已排序的子序列，得到完整的有序序列

■ 归并排序的过程描述：

➤ MERGE-SORT(A, p, r)

➤ MERGE(A, p, q, r)

详见2.3, P16~22

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

◆ 分治与递归

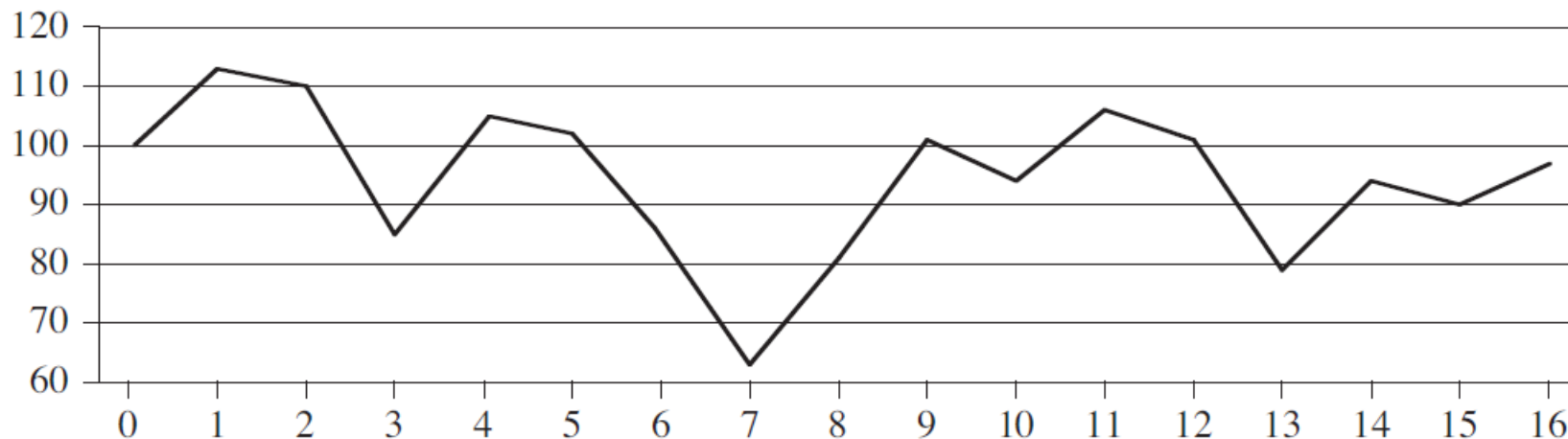
由于分解出来的子问题的性质与原问题一样，所以对子问题的求解实际上是算法的递归执行。

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，就不需要再进一步分解了，这种情况称为**基本情况**(base case).
 - **基本情况的子问题可以直接求解。**
- 若子问题的规模还比较大，需要进一步分解并递归求解，这种情况称为**递归情况**(recursive case)。

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

■ 问：哪段时间最赚钱？

即股市有起落，从哪天到哪天的收益最大呢？

■ 从问题定义到建模求解

求解炒股问题的算法模型：**最大子数组问题**

已知数组A，在A中寻找“**和**”最大的**非空连续子数组**。

	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

——称这样的连续子数组为**最大子数组**(*maximum subarray*)

■ 怎么求解？

方法一：暴力求解法 (brute-force solution)

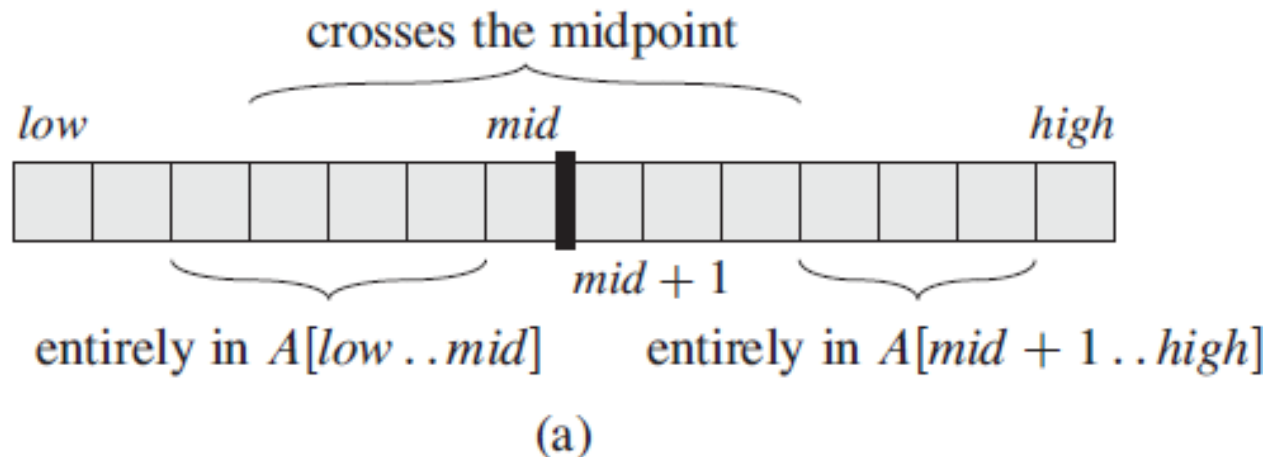
时间复杂度： $\binom{n-1}{2} = \Theta(n^2)$

方法二：使用分治策略求解

设当前要寻找子数组 $A[\text{low} \dots \text{high}]$ 的最大子数组。

分治的基本思想是：

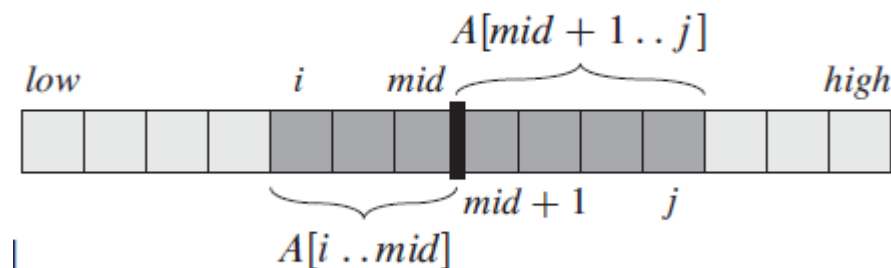
- ◆ 首先将子数组 $A[\text{low} \dots \text{high}]$ 划分为两个规模尽量相等的子数组
- ◆ 然后分别求解 $A[\text{low} \dots \text{mid}]$ 和 $A[\text{mid} + 1 \dots \text{high}]$ 的最大子数组。



基于上述划分有：

$A[\text{low} \dots \text{high}]$ 的连续子数组 $A[i \dots j]$ 所处的位置必是下面三种情况之一：

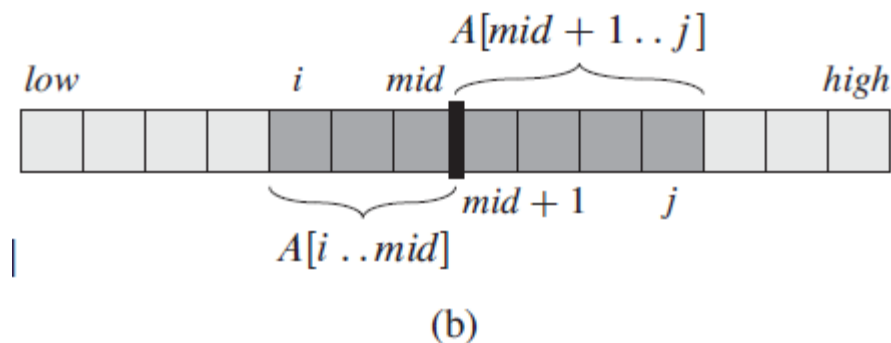
- entirely in the subarray $A[\text{low} \dots \text{mid}]$, so that $\text{low} \leq i \leq j \leq \text{mid}$,
- entirely in the subarray $A[\text{mid} + 1 \dots \text{high}]$, so that $\text{mid} < i \leq j \leq \text{high}$, or
- crossing the midpoint, so that $\text{low} \leq i \leq \text{mid} < j \leq \text{high}$.



(b)

$A[\text{low}..\text{high}]$ 的一个**最大子数组**所处的位置也必然是这三种情况之一。

即： $A[\text{low}..\text{high}]$ 的这个“最大子数组”必然是：或者完全位于 $A[\text{low}.. \text{mid}]$ 中、或者完全位于 $A[\text{mid}+1 .. \text{high}]$ 中、或者是跨越中点的所有子数组中和最大的那个。

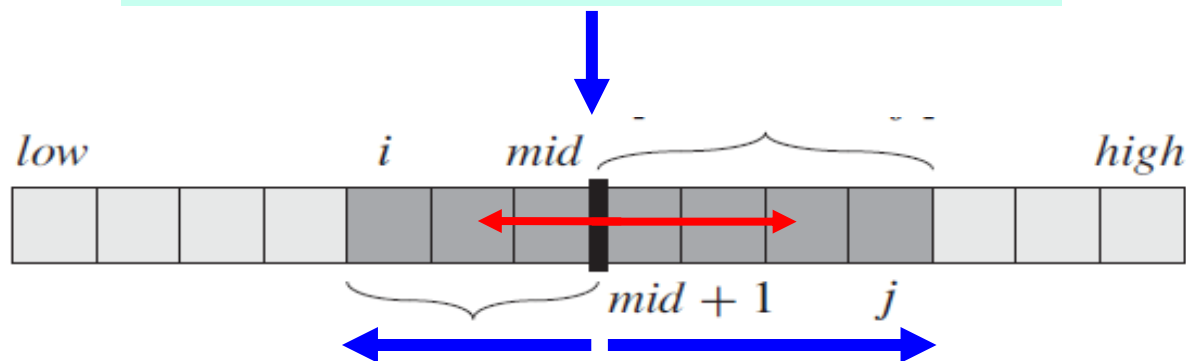


求解过程分析

1) 对于完全位于 $A[\text{low} \dots \text{mid}]$ 和 $A[\text{mid} + 1 \dots \text{high}]$ 中的最大子数组，可以在这两个较小的子数组上用递归的方法进行求解。

2) 怎么寻找跨越中点的最大子数组呢？

这样的子数组必然跨越中点 mid



从 mid 出发，分别向左和向右找出和最大的子区间，然后合并这两个区间即可得到跨越中点时的 $A[\text{low} \dots \text{high}]$ 的最大子数组。

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

FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)

```
1   $left-sum = -\infty$ 
2   $sum = 0$ 
3  for  $i = mid$  downto  $low$ 
4       $sum = sum + A[i]$ 
5      if  $sum > left-sum$ 
6           $left-sum = sum$ 
7           $max-left = i$ 
8   $right-sum = -\infty$ 
9   $sum = 0$ 
10 for  $j = mid + 1$  to  $high$ 
11      $sum = sum + A[j]$ 
12     if  $sum > right-sum$ 
13          $right-sum = sum$ 
14          $max-right = j$ 
15 return ( $max-left, max-right, left-sum + right-sum$ )
```

左侧最大的连续子数组的和

右侧最大的连续子数组的和

返回搜索的结果

◆ 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($A, low, high$)

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )           // base case: only one element
3  else  $mid = \lfloor (low + high) / 2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
          FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
5      ( $right-low, right-high, right-sum$ ) =
          FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
6      ( $cross-low, cross-high, cross-sum$ ) =
          FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
7      if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
8          return ( $left-low, left-high, left-sum$ )
9      elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
10         return ( $right-low, right-high, right-sum$ )
11     else return ( $cross-low, cross-high, cross-sum$ )
```

返回三个最大子
子数组中的大者
作为问题的解。

FIND-MAXIMUM-SUBARRAY的时间分析

令 $T(n)$ 表示求解 n 个元素的最大子数组问题的执行时间。

- 1) 当 $n=1$ 时, $T(1)=\Theta(1)$ 。否则,
- 2) 每个子问题的时间为 $T(n/2)$, 两个子问题递归求解的总时间是 $2T(n/2)$ 。
- 3) FIND-MAX-CROSSING-SUBARRAY 的时间是 $\Theta(n)$ 。

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

4.2 Strassen矩阵乘法

回顾一下矩阵运算

已知两个 n 阶方阵： $A = (a_{ij})_{n \times n}$ ， $B = (b_{ij})_{n \times n}$

1) 矩阵加法

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

时间复杂度： $\Theta(n^2)$

2) 矩阵乘法

$$C = AB = (c_{ij})_{n \times n} \text{ , 其中, } c_{ij} = \sum_{1 \leq k \leq n} a_{ik} b_{kj}, i, j = 1, 2, \dots, n$$

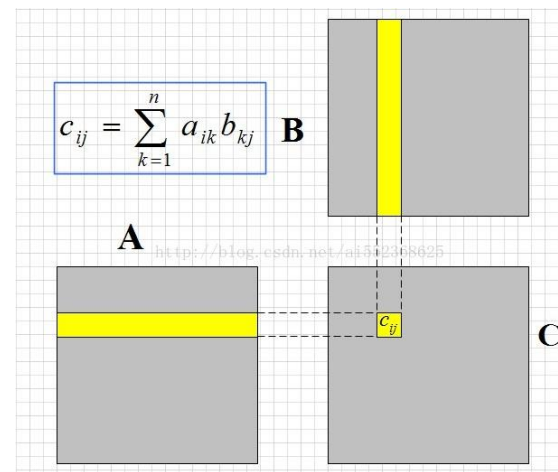
时间复杂度： $\Theta(n^3)$ 。

实现两个 $n \times n$ 矩阵乘的过程

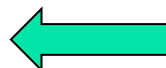
朴素的矩阵乘

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = \underline{c_{ij} + a_{ik} \cdot b_{kj}}$ 
8  return  $C$ 
```



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



朴素矩阵乘的计算时间是 $\Theta(n^3)$.

能否可以用少于 $\Theta(n^3)$ 的时间完成矩阵乘的计算?

1969年德国数学家Strassen: $n^{2.81}$

Strassen矩阵乘法：基于分治策略的矩阵乘算法

2X2的两个矩阵相乘

1) 直接相乘

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

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

2) Strassen的计算方法:

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

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

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

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

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

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

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

则,

$$\begin{aligned} c_{11} &= P + S - T + V = (a_{11} + a_{22})(b_{11} + b_{22}) + a_{22} (b_{21} - b_{11}) \\ &\quad - (a_{11} + a_{12}) b_{22} + (a_{12} - a_{22}) (b_{21} + b_{22}) \\ &\equiv a_{11} b_{11} + a_{12} b_{21} \end{aligned}$$

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

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

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

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

分析:

- 乘法次数: 7次
- 加(减)法次数: 18次

特点:

- 增加了加(减)法计算量, 减少了乘法计算量

矩阵乘的分治思路

- 设 $n = 2^k$ ，两个 n 阶方阵为

$$A = (a_{ij})_{n \times n}$$


$$B = (b_{ij})_{n \times n}$$

(注：若 $n \neq 2^k$ ，可通过在 A 和 B 中补 0 使之变成阶是 2 的幂的方阵)

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

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

方法1：朴素的分治思想 —— 简单的矩阵分块相乘


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

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

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

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

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

共有：8次 $(n/2) \times (n/2)$ 矩阵乘
4次 $(n/2) \times (n/2)$ 矩阵加

注：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 $(n/2) \times (n/2)$ 矩阵乘
- 2) 4次 $(n/2) \times (n/2)$ 矩阵加

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

其中， b , d 是常数

$$T(n) = O(n^3)$$

方法2: Strassen矩阵乘的一般方法

令

$$\mathbf{P} = (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22})$$

$$\mathbf{Q} = (\mathbf{A}_{21} + \mathbf{A}_{22}) \mathbf{B}_{11}$$

$$\mathbf{R} = \mathbf{A}_{11} (\mathbf{B}_{12} - \mathbf{B}_{22})$$

$$\mathbf{S} = \mathbf{A}_{12} (\mathbf{B}_{21} - \mathbf{B}_{11})$$

$$\mathbf{T} = (\mathbf{A}_{11} + \mathbf{A}_{12}) \mathbf{B}_{22}$$

$$\mathbf{U} = (\mathbf{A}_{11} - \mathbf{A}_{21}) (\mathbf{B}_{11} + \mathbf{B}_{12})$$

$$\mathbf{V} = (\mathbf{A}_{12} - \mathbf{A}_{22}) (\mathbf{B}_{21} + \mathbf{B}_{22})$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

则,

$$\mathbf{C}_{11} = \mathbf{P} + \mathbf{S} - \mathbf{T} + \mathbf{V}$$

$$\mathbf{C}_{12} = \mathbf{R} + \mathbf{T}$$

$$\mathbf{C}_{21} = \mathbf{Q} + \mathbf{S}$$

$$\mathbf{C}_{22} = \mathbf{P} + \mathbf{R} - \mathbf{Q} - \mathbf{U}$$

注: Strassen矩阵乘也是一个递归求解过程。

Strassen矩阵乘法的计算复杂度分析

令 $T(n)$ 表示两个 $n=2^k$ 阶矩阵的Strassen矩阵乘所需的计算

时间，则有：

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \quad \text{其中, } a \text{ 和 } b \text{ 是常数}$$

化简： $T(n) = an^2(1 + 7/4 + (7/4)^2 + \dots + (7/4)^{k-1}) + 7^k T(1)$

$$\leq cn^2(7/4)^{\log n} + 7^{\log n}$$

$$= cn^2 n^{\log(7/4)} + n^{\log 7}$$

$$= cn^{\log 4 + \log 7 - \log 4} + n^{\log 7}$$

$$= (c+1)n^{\log 7}$$

$$= O(n^{\log 7}) \approx O(n^{2.81})$$

Strassen矩阵乘法的应用场景

性能分析：

矩阵大小	朴素矩阵算法 (秒)	Strassen算法 (秒)
32	0.003	0.003
64	0.004	0.004
128	0.021	0.071
256	0.09	0.854
512	0.782	6.408
1024	8.908	52.391

- 1) 采用Strassen算法作递归运算，需要创建大量的动态二维数组，其中分配堆内存空间将占用大量计算时间，从而掩盖了Strassen算法的优势。
- 2) 于是对Strassen算法做出改进，设定一个界限。当 $n < \text{界限}$ 时，使用普通法计算矩阵，而不继续分治递归。需要合理设置界限，不同环境（硬件配置）下界限不同。
- 3) 矩阵乘法一般意义上还是选择的是朴素的方法，只有当矩阵变稠密，而且矩阵的阶数很大时，才会考虑使用Strassen算法。

4.3 求解递归式

分治和递归是“一对好兄弟”。

设原始问题的规模为 n ，之后被分解为两个子问题，子问题的规模分别 n_1 和 n_2 。

用 $T(n)$ 表示对规模为 n 的问题进行求解的时间，则规模分别为 n_1 和 n_2 的子问题的求解时间可表示为 $T(n_1)$ 和 $T(n_2)$ 。

◆ 一般, $T(n)$ 和 $T(n_1)$ 、 $T(n_2)$ 的关系可表示为

$$T(n) = T(n_1) + T(n_2) + \underline{f(n)}$$

计算过程中, 除递归求解子问题以外, 其它必要处理所花费的时间

◆ 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: $T(n) = 2T(n/2) + cn$

或如 二分查找: $T(n) = T(n/2) + 1$

- ◆ 分治算法的计算时间表达式往往是递归式。
- ◆ 那么如何化简递归式，以得到形式简单的限界函数？

介绍三种常用的递归式求解方法：

- 代换法
- 递归树法
- 主方法

注：递归式求解的目标是得到形式简单的渐近限界函数表示
(即用 O 、 Ω 、 Θ 表示的函数式)。

预处理——对表达式细节的简化

为便于处理，通常做如下假设和简化处理

(1) 运行时间函数 $T(n)$ 的定义中，一般假定自变量 n 为正整数。

➤ 因为这样的 n 通常表示数据的个数。

(2) 忽略递归式的边界条件，即 n 较小时函数值的表示。

➤ 原因在于，虽然递归式的解会随着 $T(1)$ 值的改变而改变，但此改变不会超过常数因子，对函数的阶没有根本影响。

(3) 对上取整、下取整运算做合理简化

如：

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

通常忽略上、下取整函数，就可写作以下简单形式：

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

1) 代换法(The substitution method for solving recurrences)

用代换法解递归式基本思想：

先猜测解的形式，然后用数学归纳法求出解中的常数，并证明解是正确的。

- 用代换法解递归式的步骤：

- (1) 猜测解的形式

- (2) 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

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

怎么猜？

该式与 $T(n) = 2T(n/2) + n$ 类似，故猜测其解为

$$T(n) = O(n \log n)$$

代入法要证明的是：如何恰当选择常数 c ，使得 $T(n) \leq cn \log n$ 成立？

所以现在设法证明： $T(n) \leq cn \log n$ ，并**确定常数 c 的存在。**

证明：

假设该界对 $\lfloor n/2 \rfloor$ 成立，即 $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$ ，

则在数学归纳法推论证明阶段对递归式做代换，有：

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

$$\leq cn \log(n/2) + n$$

去掉底函数

$$= cn \log n - cn \log 2 + n$$

对数运算

$$= \underline{cn \log n - (c - 1)n}$$

化简结果

故，要使 $T(n) \leq cn \log n$ 成立，只要 $c \geq 1$ 就可以，这样的 c 是存在的、合理的。

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： $T(n) \leq cn \log n$ 的结论对于较小的n成立吗？

分析：事实上，对 $n=1$ ，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 $n=1$ ，带入表达式有： $T(1) \leq c \cdot 1 \cdot \log 1 = 0$ ，
与 $T(1)=1$ 不相符。

怎么处理？

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) 依然合理地假设 $T(1) = 1$ 。

(2) 研究什么样的 c 使得 $T(2)$ 、 $T(3)$ 可以满足 $T(n) \leq cn \log n$ 。

(即使得 $T(2) \leq 2c \log 2$ 且 $T(3) \leq 3c \log 3$ 成立)

- ◆ 将 $T(1)=1$ 带入递归式，有： $T(2) = 4$ ， $T(3)=5$
- ◆ 要使 $T(2) \leq 2c \log 2$ 和 $T(3) \leq 3c \log 3$ 成立，只要 $c \geq 2$ 即可。
- ◆ 综上所述，取常数 $c \geq 2$ ，结论 $T(n) \leq cn \log n$ 成立。
命题得证。

如何猜测递归式的解呢?

1) 主要靠经验

- ◆ 尝试1：看有没有形式上类似的表达式，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 $T(n) = 2T(\lfloor n/2 \rfloor) + n$ 猜测有 $T(n) = O(n)$

似乎有 $T(n) \leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n = O(n)$ 成立

但是错误，原因：并未证出一般形式 $T(n) \leq cn$ 成立 ($cn + n \not\leq cn$)

必要的时候要做一些技术处理

1) 去掉一个低阶项：见书上的例子

2) **变量代换**：对陌生的递归式做些简单的**代数变换**，使之变成较熟悉的形式。

例：设有递归式 $T(n) \leq 2T(\lfloor \sqrt{n} \rfloor) + \log n$

分析：原始形态比较复杂

(1) **做代数代换**：令 $m = \log n$ ，则 $n = 2^m$ ，

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

得：

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

再设 $S(m) = T(2^m)$ ，得以下形式递归式：

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

从而获得形式上熟悉的递归式。

可得新的递归式的上界是：

$$O(m \log m)$$

再将 $S(m)$ 、 $m = \log n$ 带回 $T(n)$ ，有，

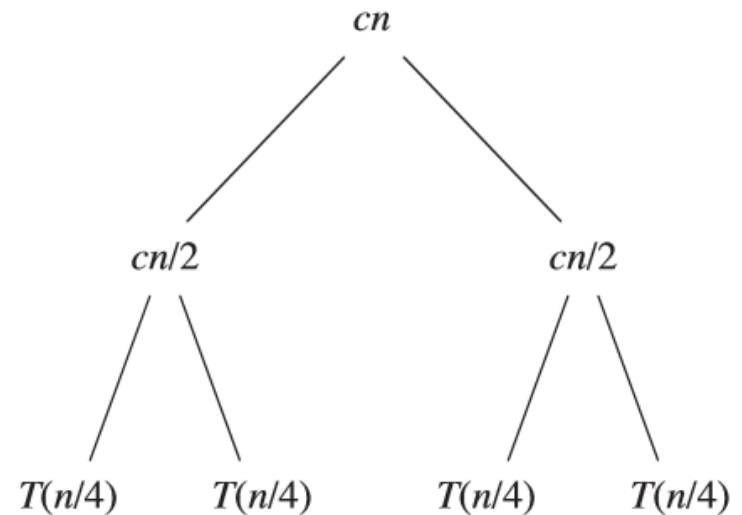
$$\begin{aligned} T(n) &= T(2^m) \\ &= S(m) = O(m \log m) \\ &= O(\log n \log \log n) \end{aligned}$$

这里， $m = \log n$

2) 递归树法(The recursion-tree method for solving recurrences)

- 根据递归式的定义，可以画一棵递归树
- 递归树：反应递归的执行过程。每个节点表示一个单一子问题的代价，子问题对应某次递归调用。根节点代表顶层调用的代价，子节点分别代表各层递归调用的代价。

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



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

准备性工作：

(1) 去掉底函数的表示

- 理由：底函数和顶函数对递归式求解并不“重要”。

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

- 一般，当证明 $n=4^k$ 成立后，再加以适当推广，就可以把结论推广到 n 不是4的幂的一般情况了。

(3) 展开 $\Theta(n^2)$ ，代表递归式中非重要项。

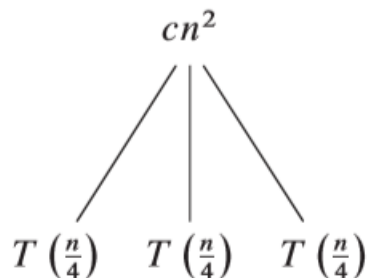
- 假设其常系数为 c ， $c>0$ ，从而去掉 Θ 符号，转变成 cn^2 的形式，便于后续的公式化简。

最终得以下形式的递归式： $T(n) = 3T(n/4) + cn^2$

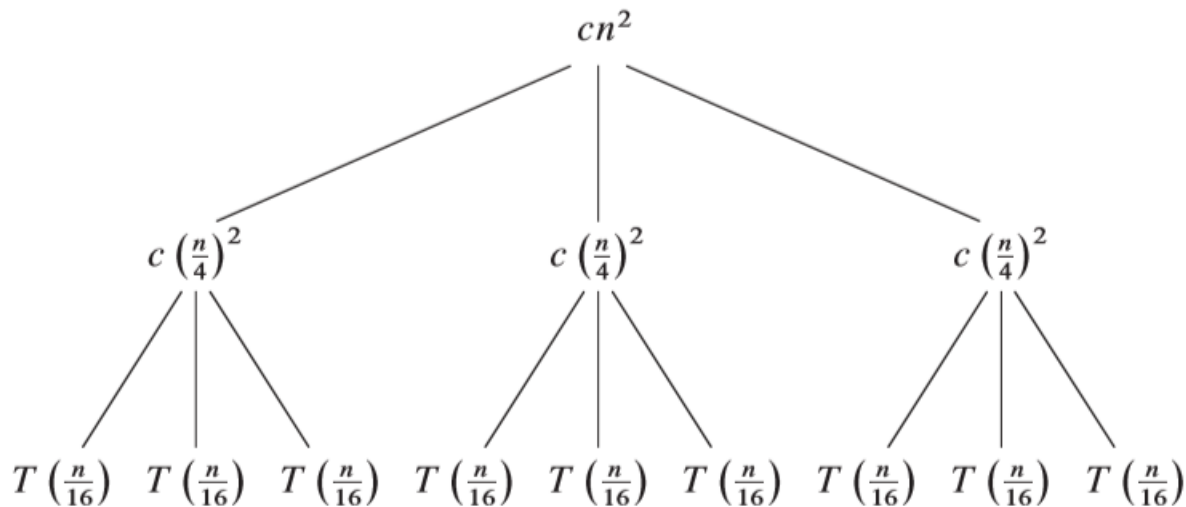
$$T(n) = 3T(n/4) + cn^2$$

用递归树描述 $T(n)$ 的演化过程:

$T(n)$



(a)



(b)

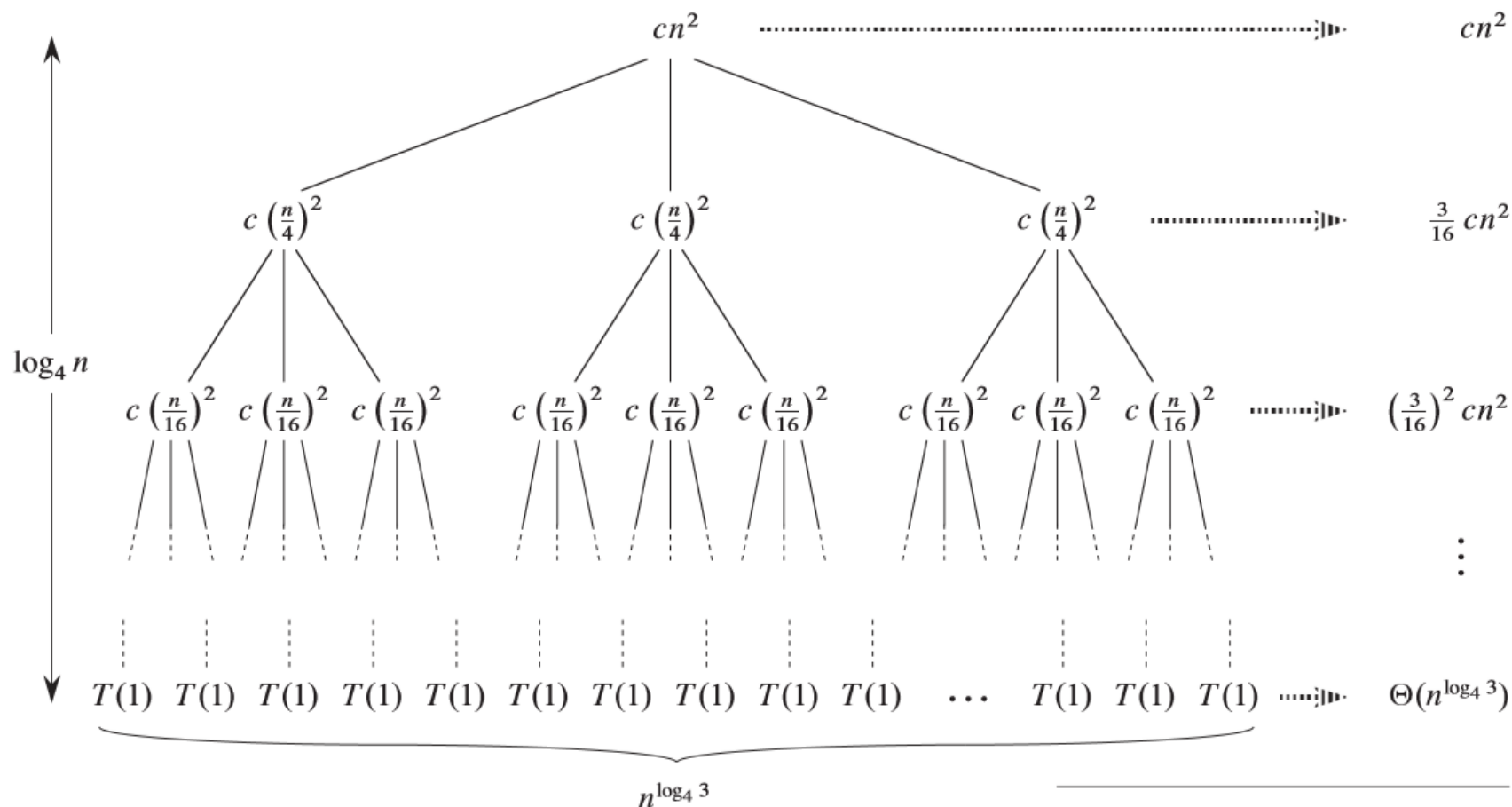
(c)

a) 对原始问题 $T(n)$ 的描述。

b) 第一层递归调用的分解情况， cn^2 是顶层计算除递归以外的代价， $T(n/4)$ 是分解出来的规模为 $n/4$ 的子问题的代价，总代价 $T(n)=3T(n/4)+cn^2$ 。

c) 第二层递归调用的分解情况。 $c(n/4)^2$ 是三棵二级子树除递归以外的代价。

继续扩展下去，直到递归的最底层，得到如下形式的递归树：



(d)

Total: $O(n^2)$

d) 完全扩展的递归树，**递归树高度为 $\log_4 n$ (共有 $\log_4 n + 1$ 层)**

树的深度：子问题的规模按 $1/4$ 的方式减小，在递归树中，深度为 i 的节点，子问题的大小为 $n/4^i$ 。

当 $n/4^i=1$ 时，子问题规模仅为1，达到边界值。

所以，

- 节点分布层： $0 \sim \log_4 n$
- 树共有 $\log_4 n + 1$ 层
- 从第2层起，每一层上的节点数为上层节点数的3倍
- 深度为 i 的层节点数为 3^i 。

代价计算

(1) **内部节点**: 位于 $0 \sim \log_4 n - 1$ 层

深度为 i 的节点的局部代价为 $c(n/4^i)^2$,

i 层节点的总代价为: $3^i c(n/4^i)^2 = (3/16)^i cn^2$ 。

(2) **叶子节点**: 位于 $\log_4 n$ 层, 共有 $3^{\log_4 n} = n^{\log_4 3}$ 个,

每个叶子节点的代价为 $T(1)$,

叶子节点总代价为 $n^{\log_4 3} T(1) = \Theta(n^{\log_4 3})$

(3) 树的总代价


整棵树的总代价等于各层代价之和，则有

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

利用等比数列化简上式。

T(n)中， cn^2 项的系数构成一个递减的几何级数。

将T(n)扩展到无穷，即有


$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) \end{aligned}$$

至此，获得T(n)解的一个猜测： $T(n)=O(n^2)$ ，成立吗？

用代换法证明猜测的正确:

- 将 $T(n) \leq dn^2$ 作为归纳假设, d 是待确定的常数, 带入推论证明过程, 有

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \\ &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \end{aligned}$$

显然, 要使得 $T(n) \leq dn^2$ 成立, 只要 $d \geq (16/13)c$ 即可。所以,
 $T(n) \leq dn^2$ 的猜测成立。

定理得证 (边界条件的讨论略)。

另: $O(n^2)$ 是 $T(n)$ 的一个紧确界, 为什么?

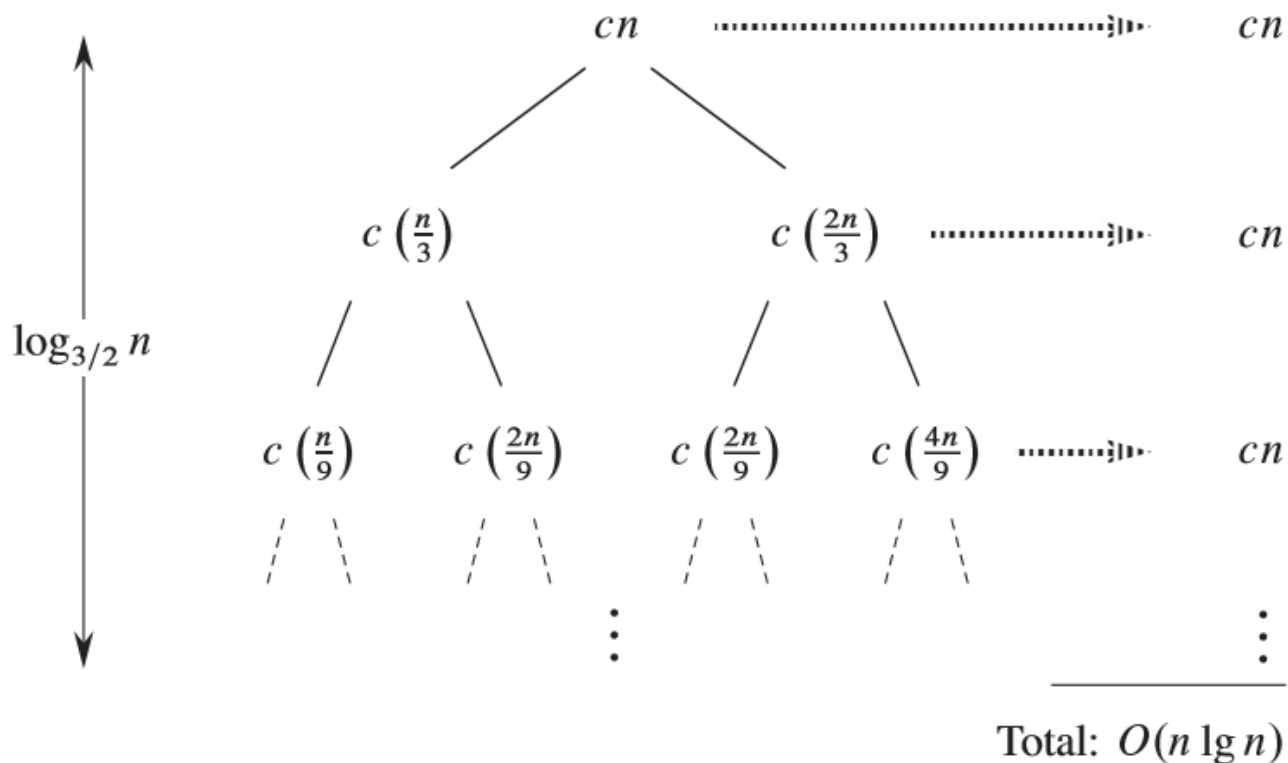
例 求表达式 $T(n) = T(n/3) + T(2n/3) + O(n)$ 的上界

(这里, 表达式中直接省略了下取整和上取整函数)

进一步地, **引入常数c**, 展开 $O(n)$, 得:

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

递归树为:



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都是 cn ，而是 $\leq cn$ 的某个值。

- **树的深度：**

- 在上述形态中，最长的路径是最右侧路径，由

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

组成。

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

■ 递归式解的猜测：

- 至此，我们可以合理地猜测该树的总代价**至多是层数乘以每层的代价**，并鉴于上面关于层代价的讨论，我们可以假设递归式的上界为：

$$O(c n \log_{3/2} n) = O(n \log n)$$

注：这里，我们假设每层的代价为 cn 。

事实上， cn 为每层代价的上界，这一假设是合理的细节简化处理。

猜测的证明：证明 $O(n\log n)$ 是递归式的上界

即证明： **$T(n) \leq dn \log n$** ,

d 是待确定的合适的正常数

$$\begin{aligned} T(n) &\leq T(n/3) + T(2n/3) + cn \\ &\leq d(n/3) \log(n/3) + d(2n/3) \log(2n/3) + cn \\ &= (d(n/3) \log n - d(n/3) \log 3) + (d(2n/3) \log n - d(2n/3) \log 3/2) + cn \\ &= dn \log n - d((n/3) \log 3 + (2n/3) \log(3/2)) + cn \\ &= dn \log n - d((n/3) \log 3 + (2n/3) \log 3 - (2n/3) \log 2) + cn \\ &= dn \log n - \underline{dn(\log 3 - 2/3)} + cn \leq dn \log n \end{aligned}$$

上式的成立条件： $d \geq c / (\log 3 - (2/3))$

\therefore 猜测正确，递归式解得证。


3) 主方法 (The master method for solving recurrences)



如果递归式有如下形式，在满足一定的条件下，可以用**主方法**直接给出渐近界：

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

其中， a 、 b 是常数，且 $a \geq 1$ ， $b > 1$ ； $f(n)$ 是一个渐近正的函数。



注：这里采用了细节的简化，没有考虑 n/b 的取整问题，省略了下取整、上取整，但本质上不影响对递归式渐近行为的分析。

对上述形式的递归式渐近界的求解可用称之为“主定理”的结论给出的。

定理2.1 主定理

设 $a \geq 1$ 和 $b > 1$ 为常数，设 $f(n)$ 为一函数， $T(n)$ 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 $T(n)$ 可能有如下的渐近界：

- 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 $c < 1$ 与所有足够大的 n ，有 $af(n/b) \leq cf(n)$ ，则 $T(n) = \Theta(f(n))$ 。

理解主定理:

1) $T(n)$ 的解似乎与 $f(n)$ 和 $n^{\log_b a}$ 有“密切关联”:

$f(n)$ 和 $n^{\log_b a}$ 比较, $T(n)$ 取了其中较大的一个。

如: 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, 则乘以对数因子, 得

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

2) 在第一种情况中, $f(n)$ 要**多项式**地小于 $n^{\log_b a}$ 。即, 对某个常量 $\varepsilon > 0$, $f(n)$ 必须渐近地小于 $n^{\log_b a}$, 两者相差了一个 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 $n^{\log_b a}$, 还要满足一个“规则性”条件 $af(n/b) \leq cf(n)$ 。

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

- ◆ $f(n)$ 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。

使用主方法：分析递归式满足主定理的哪种情形，即可得到解（无需证明）。

例2.6 解递归式 $T(n) = 9T(n/3) + n$

分析：这里， $a=9$ ， $b=3$ ， $f(n)=n$ 。

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

因为 $f(n) = n = O(n^{\log_3 9 - \varepsilon})$ ，其中取 $\varepsilon=1$ ，

所以对应主定理的第一种情况。

于是有： $T(n) = O(n^2)$

例2.7 解递归式 $T(n) = T(2n/3) + 1$

分析：这里， $a=1$ ， $b=3/2$ ， $f(n)=1$ ，因此有

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

且有

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

故主定理第二种情况成立，即 $T(n) = \Theta(\log n)$

例2.8 解递归式 $T(n) = 3T(n/4) + n \log n$

分析：这里， $a=3$ ， $b=4$ ， $f(n)=n \log n$ ，

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

故， $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ 成立，其中可取 $\varepsilon \approx 0.2$ 。

同时，对足够大的 n ，

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

其中， **$c = 3/4$** 。

所以第三种情况成立， **$T(n) = \Theta(n \log n)$** 。

例2.9 递归式 $T(n) = 2T(n/2) + n \log n$ 不能用主方法求解

分析：这里， $a=2$ ， $b=2$ ，

$$n^{\log_b a} = n^{\log_2 2} = O(n)$$

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

第三种情况成立吗？

事实上不成立，因为对于任意正常数 ε ，

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

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

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

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。



4.6 证明主定理

为什么主定理是正确的？

主定理证明：（略，P55）

注：在使用主定理时不用再证明其正确性。

还有没有其它方法化简递归式?

4) 直接化简

根据递推关系，展开递推式，找出各项系数的构造规律（如等差、等比等），最后得出化简式的最终形式。

如： $T(n) = 2T(n/2) + 2$

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

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

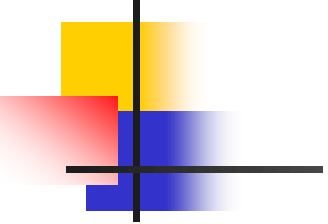
...

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

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

$$= 3n/2 - 2$$

例：化简递归式 $T(n) = 2T(n/2) + n \log n$


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

$$= 2(2T(n/4) + (n/2) \log(n/2)) + n \log n$$

$$= 2^2 T(n/2^2) + n \log n - n + n \log n$$

$$= 2^2 T(n/2^2) + 2n \log n - n$$

$$= 2^2 (2T(n/2^3) + (n/4) \log(n/4)) + 2n \log n - n$$

$$= 2^3 T(n/2^3) + n \log n - 2n + 2n \log n - n$$

$$= 2^3 T(n/2^3) + 3n \log n - 2n - n$$

$$= \dots$$

$$= 2^k T(n/2^k) + kn \log n - n \sum_{i=1}^{k-1} i$$

$$= n + kn \log n - n(k-1)k/2$$

$$= n + n \log^2 n - (n/2) \log^2 n + n \log n / 2$$

$$= O(n \log^2 n)$$

33.4 最近点对问题

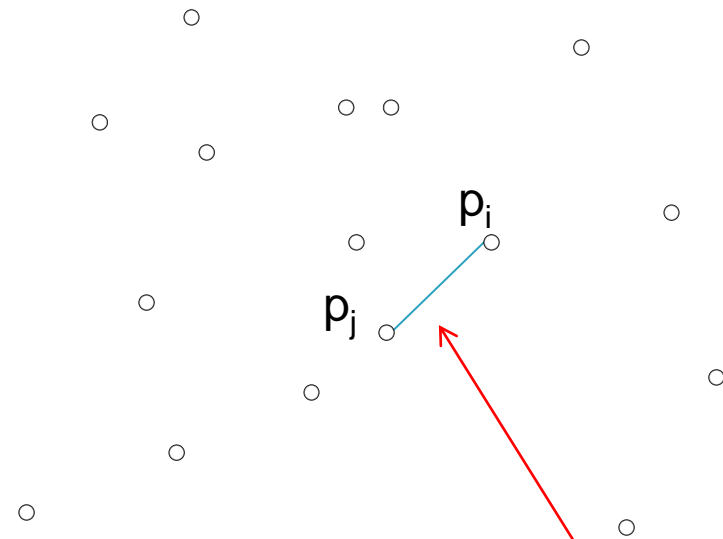
1. 问题描述

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) ， $1 \leq i \leq n$ 。两点之间的距离取其欧式距离，记为

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

问题：找出一对距离最近的点。

注：允许两个点位于同一个位置，此时两点之间的距离为0。



平面上分布的点集

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

分析：

一种方法是暴力搜索（全搜索）的方法：对每对点都计算距离，然后比较大小，找出其中的最小者。

该方法的时间复杂度为：

- 计算点间距离： $O(n^2)$ ，因为共有 $C_2^n = n(n-1)/2$ 对点间的距离要计算。
- 找最小距离： $O(n^2)$ ，因为需要 $n(n-1)/2 - 1$ 次比较。

所以，总的时间复杂度： $O(n^2)$ 。

问：有没有更好的办法？

2.求解最近点对问题的分治方法

这里，利用分治法“设计”一个具有 $O(n\log n)$ 时间复杂度的算法求解最近点对问题。

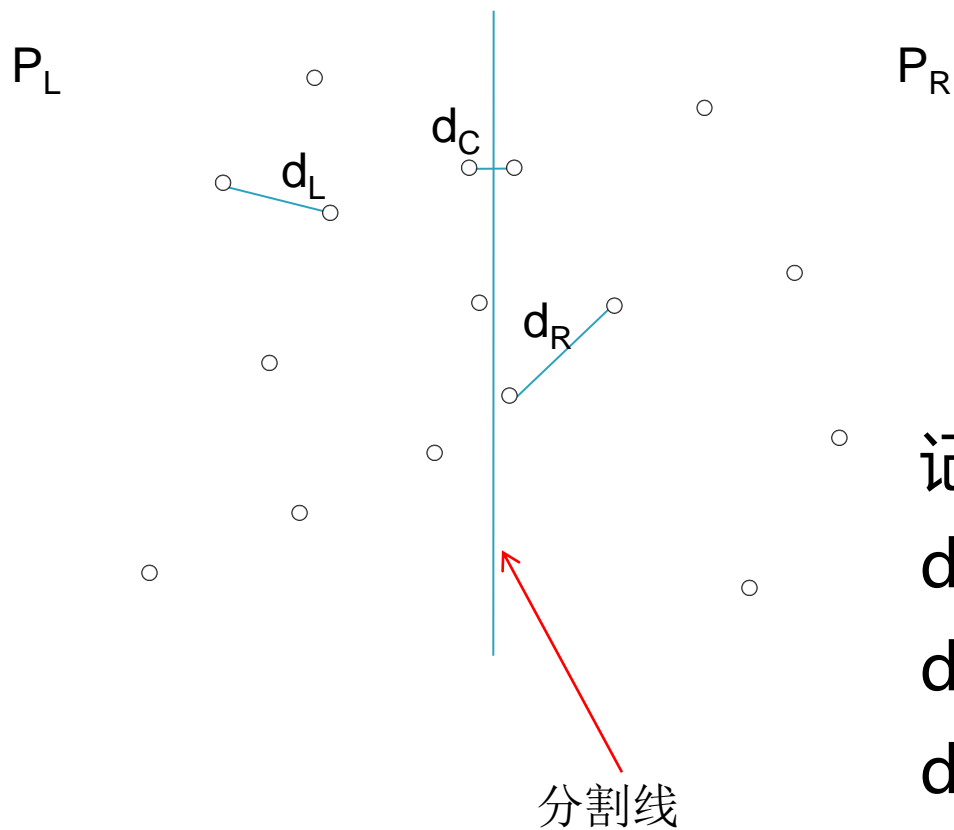
设计策略：分治

1) 首先将所有的点按照x坐标排序

排序过程需要 $O(n\log n)$ 的时间，不会从整体上增加时间复杂度的数量级 (加法规则)。

2) 划分

由于点已经按x坐标排序，所以空间上可以“想象”画一条垂线作为分割线，将平面上的点集分成左、右两半 P_L 和 P_R 。



记,

d_L : P_L 中的最近点对距离

d_R : P_R 中的最近点对距离

d_C : 跨越分割线的最近点对距离

则，最近的一对点或者在 P_L 中，或者在 P_R 中，或者一个端点在 P_L 中而另一个在 P_R 中（跨越分割线）。

建立一个递归过程求 d_L 和 d_R ，并在此基础上计算 d_C 。而且，要使得对 d_C 的计算至多只能花 $O(n)$ 的时间。

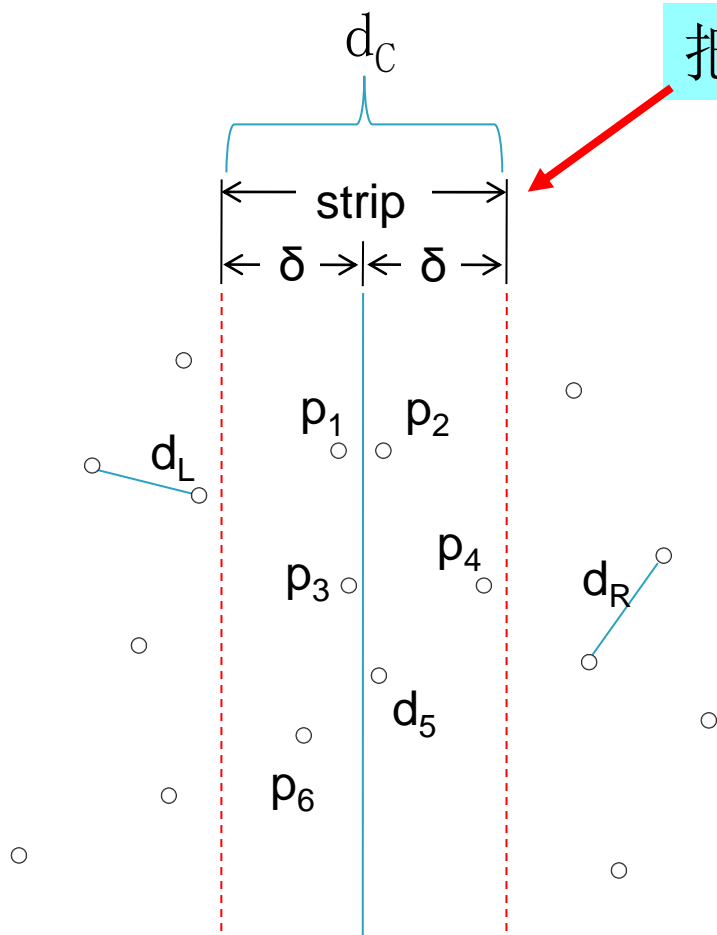
这样，递归过程将由两个大致相等的一半大小的递归调用和 $O(n)$ 附加工作组成，总的时间表示为：

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

就可以控制在 $O(n \log n)$ 以内。

令 $\delta = \min(d_L, d_R)$ ，通过观察可得：如果要满足 $d_C < \delta$ ，则 d_C 对应的点对必然在分割线两侧的 δ 距离以内。

超出带区域的 d_C 是无用的



把这个区域叫做带(strip)

3) 计算 d_c

方法一：对**均匀分布的大型稀疏点集**

- 可预计位于该带中的点均匀而“稀疏”，
- 个数平均只有 $O(\sqrt{n})$ 个点在这个带中。

因此，可以 $O(n)$ 的时间计算出这些点对之间的距离。

描述如下：

```
for  $i=1$  to numPointsInStrip do
    for  $j=i+1$  to numPointsInStrip do
        if  $\text{dist}(p_i, p_j) < \delta$ 
             $\delta = \text{dist}(p_i, p_j);$ 
```

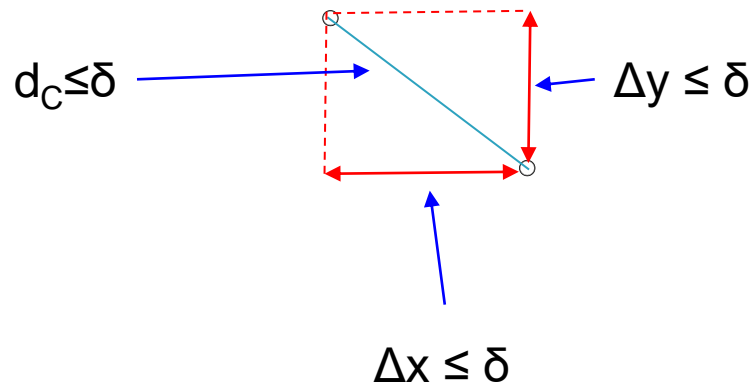
这里， $\text{numPointsInStrip} = O(\sqrt{n})$

方法一存在的问题：

在最坏的情况下，所有的点可能都在Strip内。因此，该方法不能总以线性时间运行。

如何改进？

事实上，这样的 d_C 的两个点的y坐标相差最多也不会大于 δ ，否则 $d_C > \delta$ 。



计算 d_c 的改进:

设点也按它们的 y 坐标排序, 从 p_i 开始向远处 (向下) 搜索。

假设搜索到 p_j 时, p_j 与 p_i 的 y 坐标相差大于 δ , 那么对于 p_i 而言更远的 p_j 就可以终止搜索, 转而处理 p_i 后面的点 p_{i+1} 。

算法描述如下:

for $i=1$ to numPointsInStrip do

for $j=i+1$ to numPointsInStrip do

*if p_i and p_j 's **y -coordinates differ** by more than δ*

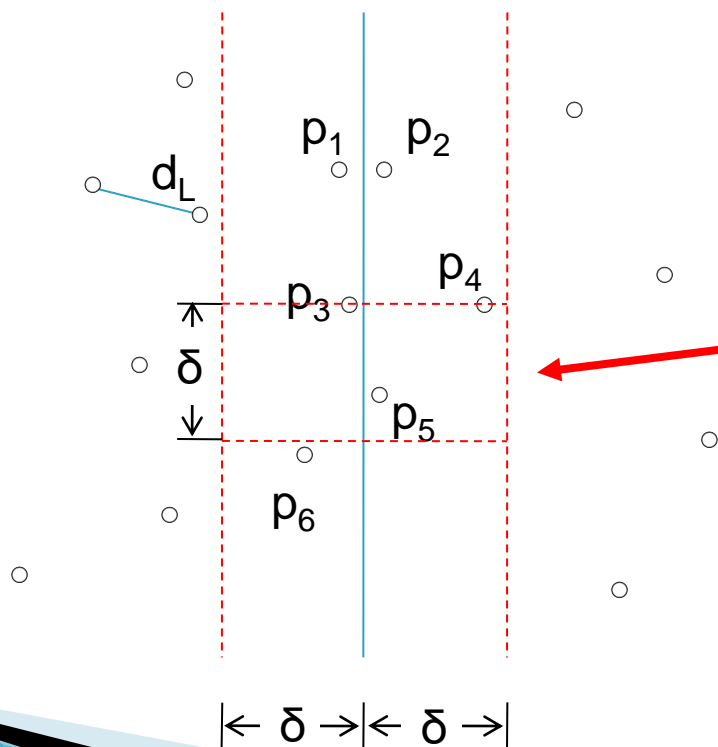
break;

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

$\delta = \text{dist}(p_i, p_j);$

分析：

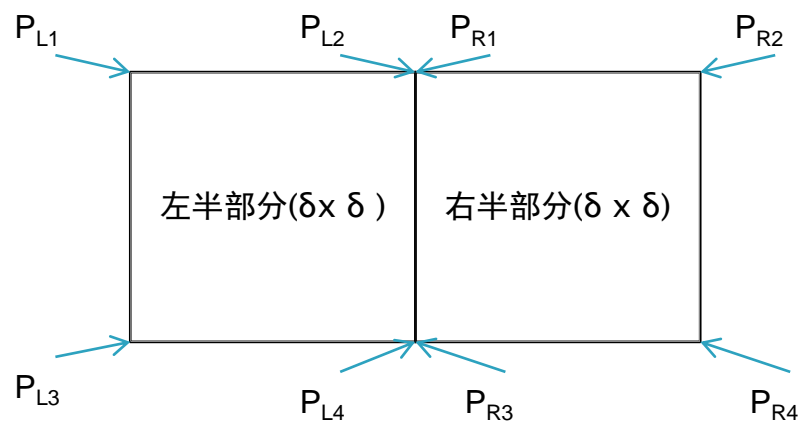
上述改进对运算时间的**影响是显著**的，因为对每一个 p_i ，在 p_i 和 p_j 的y坐标相差大于 δ 时，就会退出内层循环。这一过程中仅有少数的 p_j 被考察，大大减少了计算量。如，



对于 p_3 只有两个点 p_4 和 p_5 落在垂直距离 δ 之内的带状区域中。（对称的，只向下看就可以了）

一般情况下，对于任意的点 p_i ，在最坏情况下，
最多有7个 p_j 需要考虑。

- 每个方块最多包含4个点，其中一个点是 p_i ，另外7个就是需要考虑的 p_j 点。如图所示：



这样，对于每个 p_i ，最多有7个 p_j 要考虑，也就是**最多计算 p_i 和另外7个点的距离**，所以对每个 p_i ，计算时间可看作是 $O(1)$ 的。

则，计算 d_c 的时间就为 $O(n)$ ，即使是最坏的情况下。

于是，可得：最近点对的求解过程由两个一半大小的递归调用加上合并两个结果的线性附加工作组成：

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

y坐标的排序问题

如果每次递归都要对点的y坐标重新进行排序，则这又要有 $O(n \log n)$ 的附加工作。

总的时间为

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

若如此，整个算法的时间复杂度就为 $O(n \log^2 n)$ 。

如何处理？



解决方案：改进对点坐标进行排序的处理方法——**预排序**。

策略：(1) 设置两个表，

- ◆ **P表**：按x坐标对点排序得到的表；

- ◆ **Q表**：按y坐标对点排序得到的表。

这两个表可以在预处理阶段花费 $O(n\log n)$ 时间得到

(2) 再记， **P_L** 和 **Q_L** 是传递给左半部分递归调用的参数表，

P_R 和 **Q_R** 是传递给右半部分递归调用的参数表。

在将P分为 **P_L** 和 **P_R** 之后，复制Q到 **Q_L** 和 **Q_R** 中，然后**删除 **Q_L** 和 **Q_R** 中不在各自范围内的点**。这一操作花费 $O(n)$ 时间即可完成。而此时 **Q_L** 和 **Q_R** 均已按y坐标排好序。

然后：将 P_L 、 P_R 和经上面处理后得到的 Q_L 、 Q_R 带入递归过程进行处理， P_L 、 P_R 是按照x坐标排序的点集， Q_L 、 Q_R 是按照y坐标排序的点集。

最后：当递归调用返回时，扫描本级的Q表，**删除其x坐标不在带内的所有点**。这一处理需要 $O(n)$ 的时间。下一步，对每个 p_i ，寻找近邻中 $\Delta y \leq \delta$ 的 p_j 即可。

综上所述，所有附加工作的总时间为 $O(n)$ ，则整个算法的计算时间为

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

- ▶ 运用分治策略解决的问题一般来说具有以下特点：
- ▶ 1、原问题可以分解为多个子问题
 - 这些子问题与原问题相比，只是问题的规模有所降低，其结构和求解方法与原问题相同或相似。
- ▶ 2、原问题在分解过程中，递归地求解子问题
 - 由于递归都必须有一个终止条件，因此，当分解后的子问题规模足够小时，应能够直接求解。
- ▶ 3、在求解并得到各个子问题的解后
 - 应能够采用某种方式、方法合并或构造出原问题的解。
- ▶ 在分治策略中，由于子问题与原问题在结构和解法上的相似性，用分治方法解决的问题，大都采用了递归的形式。在各种排序方法中，如归并排序、堆排序、快速排序等，都存在有分治的思想。

作业（递归式化简部分）：

(2.4)

(4.1-5)：要求见下一页

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

(4.3-9) 利用改变变量的方法求解递归式 $T(n) = 3T(\sqrt{n}) + \lg n$ 。得到的解应是紧确的。

(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。

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

(4.5-4) 主方法能否应用于递归式 $T(n) = 4T(n/2) + n^2 \lg n$ ？为什么？给出此递归式的渐近上界。

4.1-5要求： 阅读4.1-5题面及以下程序，然后写出你对这个算法的理解。

MAX-SUBARRAY-LINEAR(A)

$n = A.length$

$max-sum = -\infty$

$ending-here-sum = -\infty$

for $j = 1$ **to** n

$ending-here-high = j$

if $ending-here-sum > 0$

$ending-here-sum = ending-here-sum + A[j]$

else $ending-here-low = j$

$ending-here-sum = A[j]$

if $ending-here-sum > max-sum$

$max-sum = ending-here-sum$

$low = ending-here-low$

$high = ending-here-high$

return ($low, high, max-sum$)