华中科技大学
计算机科学与技术学院
School of Computer Science &Technology，HUST

# 算法设计与分析


刘渝
Liu_yu@hust.edu.cn

2022秋季-华科-计算机
21级大数据

# 算法分析与设计
# 第四章
# 分治策略

# 思考：

插入排序本质上是一种什么模式的排序算法？

归并排序本质上是一种什么模式的排序算法？

两者是一样的吗？

# 结论：

插入排序的基本思路：从A[1]开始，在子数组A[1..j-1]完成排序后，将下一个元素A[j]插入到子数组A[1..j]的适当位置，从而生成一个包含更多元素的有序子数组A[1..j]。

——这种通过不断增加的策略完成计算的方法就是增量式方法。

# 归并排序　设A[1..n]是含有n个元素序列

## 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**：递归地对两个子序列进行排序，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'。

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

# 分治法

当问题规模大而无法直接求解时，将原始问题分解为几个规模较小、性质与原始问题一样的**子问题**，然后**递归地求解这些子问题**，最后合并子问题的解。

## 基本步骤

**分解**：将原问题分为若干个规模较小、相互独立，性质与原问题一样的**子问题**；

**解决**：若子问题规模较小、可直接求解时则直接解；否则“**递归**”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，然后用同一策略继续求解子问题。

**合并**：将子问题的解合并成原问题的解。

## 分治与递归

**分治的基本策略就是递归求解**

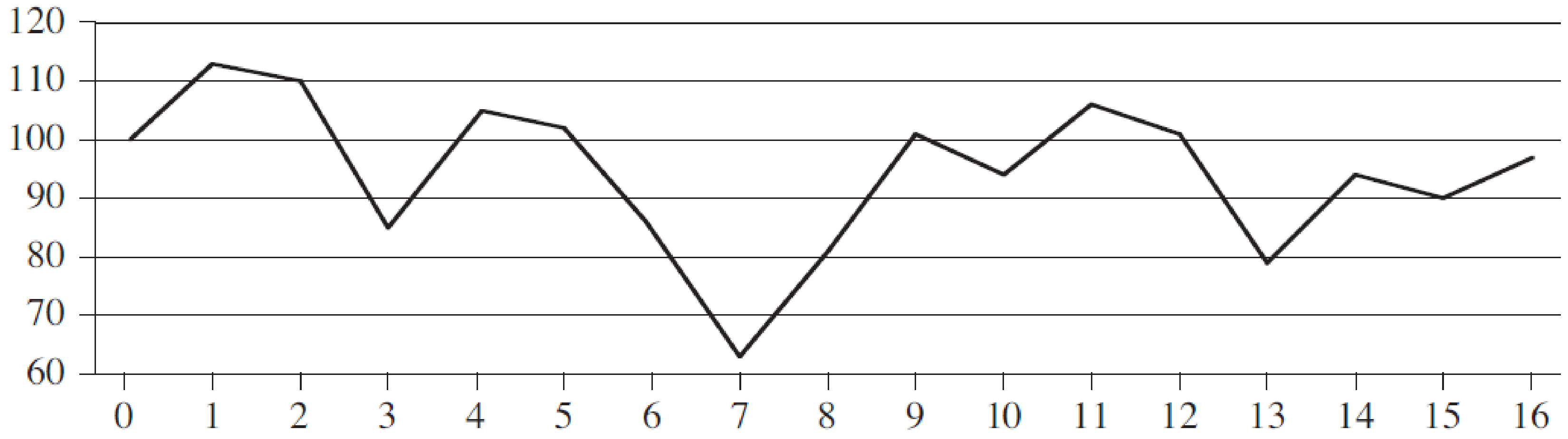
由于分解出来的子问题的性质与原问题一样，所以对子问题的求解实际上是算法的递归执行。

- 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)。

# 目　　录

# 最大子数组

## Example:炒股




| 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?**

# Example:炒股

## 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***）

# Example:炒股

## 方法一：**暴力求解**

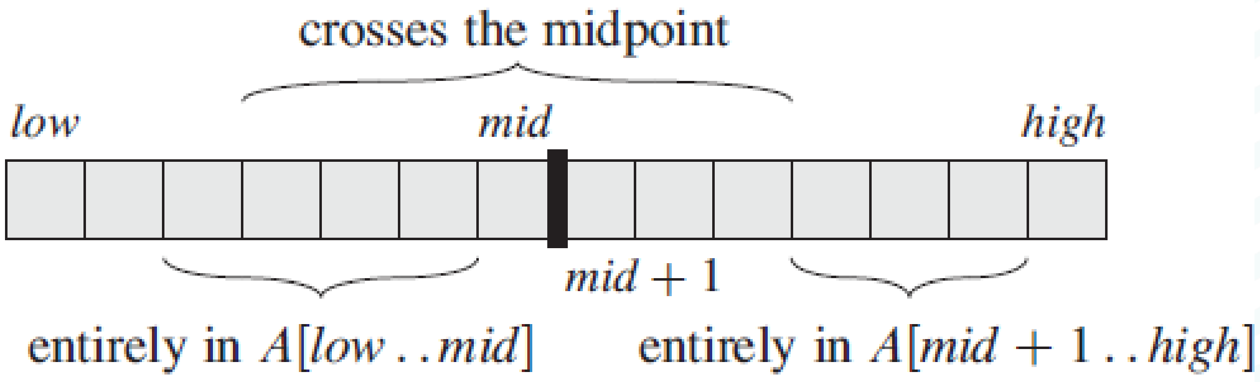
搜索A的每一对起止下标区间的和，和最大的子区间就是最大子数组，时间复杂度：

$$\binom{n-1}{2} = \Theta(n^2)$$

# Example:炒股

方法二：**分治策略**　目标：寻找子数组A[low...high]的最大子数组

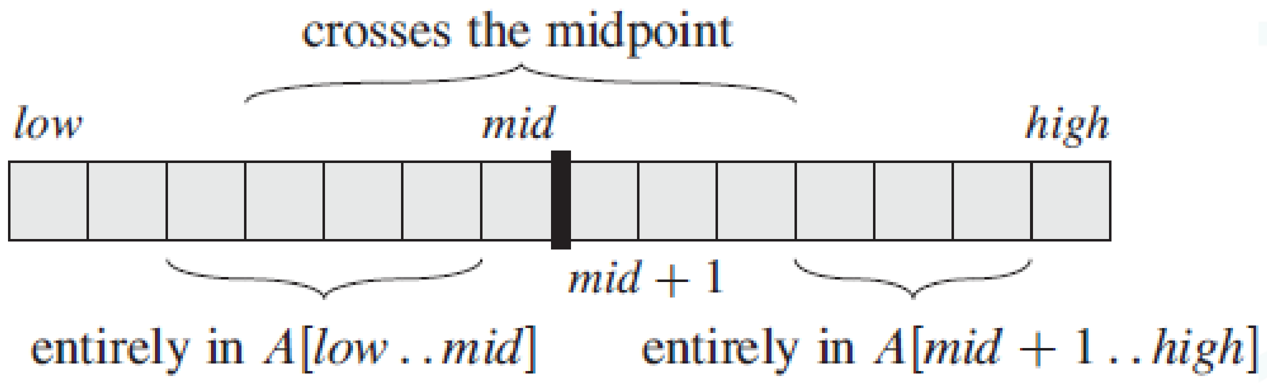
- 首先将子数组A[low...high]划分为两个规模尽量相等的子子数组，分割点：
  **mid=(low+high)/2**
- 然后分别求解A[low...mid]和A[mid+1...high]的最大子数组。




(a)

# Example:炒股

方法二：**分治策略**　　目标：寻找子数组A[low...high]的最大子数组

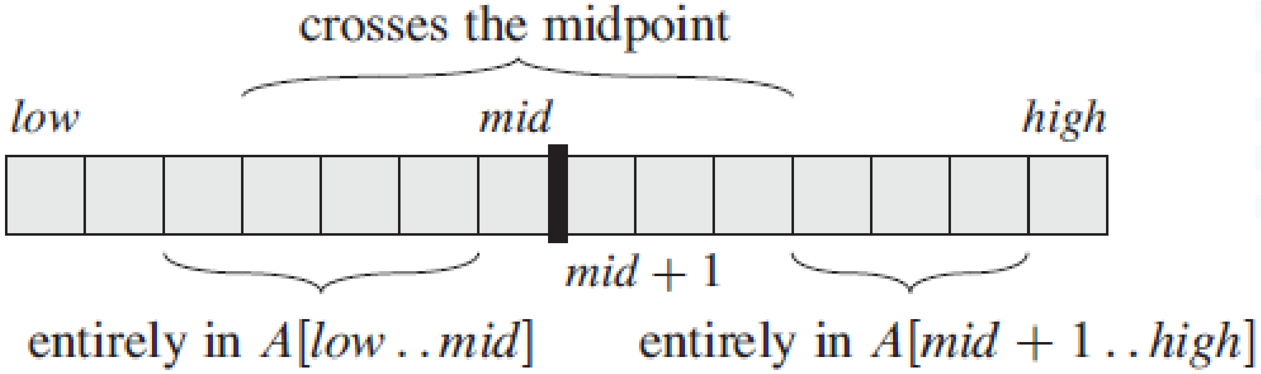



(a)

连续子数组A[i...j]所处的位置必是右面三种情况之一：

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

## 分治策略




(a)

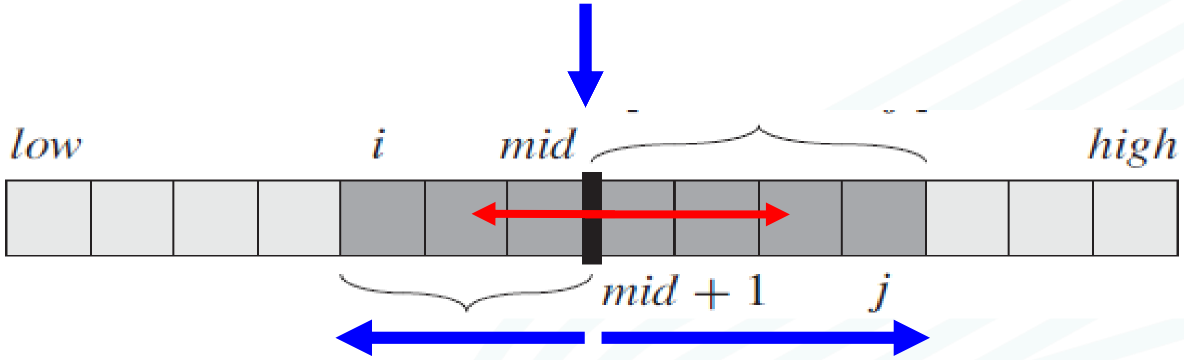
A[low..high]的最大子数组也是A[low..high]的连续子数组，所以A[low..high]的一个最大子数组所处的位置也必然是这三种情况之一。

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

# Example:炒股 求解过程

1）对于完全位于A[low .. mid]和A[mid+1.. 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）跨越中点寻找最大子数组，从mid出发，分别向左和向右找出“和最大”的子区间，mid分别是左右区间的终点和起点。然后合并这两个区间即可得到跨越中点时的A[low ... high]的最大子数组



# 求解过程

## 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

同理，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

- 时间复杂度为：$\Theta(n)$

# 求解过程

## 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)
```

求A[low~mid]的最大子数组

求A[mid+1~high]的最大子数组

求跨越中点的最大子数组

返回其中的大者作为A[low..high]的解。

# 求解过程（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)=Θ(1)**。否则，

2）对A[low..mid]和A[mid+1..high]两个子问题递归求解，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)**，**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Θ(n)。**

算法FIND-MAXIMUM-SUBARRAY执行时间T(n)的递归式为：

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

# Strassen矩阵乘法

## 矩阵运算

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

1）**矩阵加法** $C=A+B=(c_{ij})_{nxn}$，其中，

$$c_{ij}=a_{ij}+b_{ij},\ \mathrm{i,j}=1,2,\ldots,\mathrm{n}$$

时间复杂度：**Θ(n²)**

2）**矩阵乘法** $C=AB=(c_{ij})_{nxn}$，其中，

$$c_{ij}=\sum_{1\le k\le n} a_{ik}b_{kj},\ \mathrm{i,j}=1,2,\ldots,\mathrm{n}$$

时间复杂度：**Θ(n³)**。

# 矩阵相乘

```
SQUARE-MATRIX-MULTIPLY(A, B)
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
```

朴素的矩阵乘法

朴素的矩阵乘法的计算时间是Θ(n³).

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

能否用少于Θ(n³)的时间完成矩阵乘的计算?

# 基于分治策略的矩阵乘法算法

## 1）直接相乘

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

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

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

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

直接相乘共需要

8次乘法和4次加法

# 基于分治策略的矩阵乘法算法

## 2）Strassen的计算法方法

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

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

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

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

令：

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

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

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

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

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

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

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

# 基于分治策略的矩阵乘法算法

## 2）Strassen的计算法方法

令：
$P=(a_{11}+a_{22})(b_{11}+b_{22})$

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

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

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

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

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

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

$\Rightarrow$

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

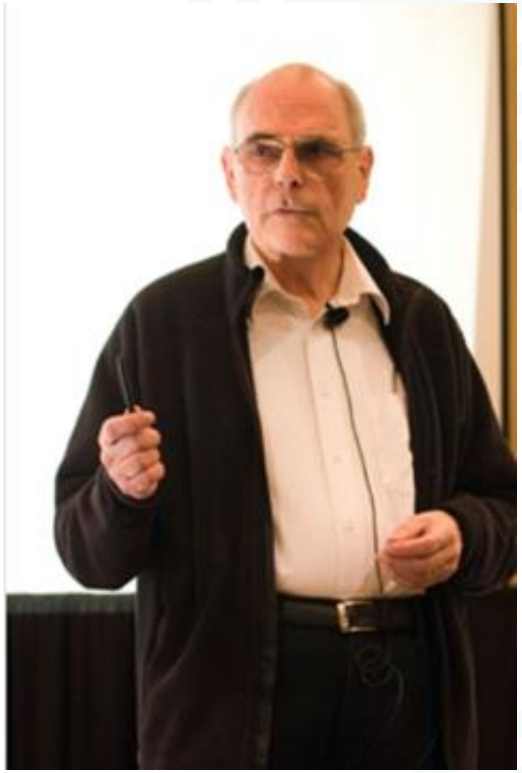
$c_{12} = R+T \equiv a_{11}b_{12}+a_{12}b_{22}$

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

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

# 基于分治策略的矩阵乘法算法

## 2）Strassen的计算方法



朴素矩阵相乘

乘法次数：8次
加（减）法次数：4次

Strassen计算方法

乘法次数：7次
加（减）法次数：18次

Strassen矩阵乘通过减少乘法计算量、适当增加加法计算量，从总体上减少矩阵乘的运算时间。

## 矩阵乘的分治思路

- 设 $n = 2^k$，两个n阶方阵为

$$A = (a_{ij})_{nxn} \qquad B = (b_{ij})_{nxn}$$

(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)x(n/2)的子矩阵：

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

# 矩阵乘的分治思路

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

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

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

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

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

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

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

共有：8次(n/2)x(n/2) 矩阵乘

4次(n/2)x(n/2) 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为止。从而构造出一个递归计算过程。

## 矩阵乘的分治思路

8次(n/2)x(n/2) 矩阵乘  4次(n/2)x(n/2) 矩阵加

令 T(n) 表示两个n×n矩阵相乘的计算时间。

则首次分块时，需要：

1) 8次(n/2)×(n/2) 矩阵乘 → 8T(n/2)

2) 4次(n/2)×(n/2) 矩阵加 → $dn^2$

故，

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

其中，b，d是常数

化简得：T(n) = $O(n^3)$

# 矩阵乘的分治思路

7次(n/2)x(n/2) 矩阵乘 18次(n/2)x(n/2) 矩阵加

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：

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

其中， a和b是常数

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

$C_{12}=R+T$

$C_{21}=Q+S$

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

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

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

注：Strassen矩阵乘也是一个递归求解过程，任意两个子矩阵块的乘可以沿用同样的规则进行。

# 矩阵乘的分治思路

T(n)表示两个$n=2^k$阶矩阵的Strassen矩阵乘所需的计算时间

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

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

$$\le an^2(7/4)^{\log n}+b7^{\log n}$$

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

$$= an^{\log 4+\log 7-\log 4} + bn^{\log 7}$$

$$= (a+b)n^{\log 7}$$

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

这里，$k=\log n$

# 矩阵乘的分治思路

- Strassen算法的发表（1969年）引起很大的轰动。
- 但从实用的角度看，Strassen算法并不是解决矩阵乘法的最好选择：

  （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×n矩阵乘的最优时间是$O(n^{2.376})$

# 求解递归式

## 分治与递归

- 分治的基本思想是递归策略
- 分治算法形式上是一个递归计算过程
- 分治算法的时间分析通常用递归关系式进行推导

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$T(n_2)$。

# 求解递归式

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

$$T(n)=\underbrace{T(n_1)+T(n_2)}_{\text{对子问题递归求解花费的时间}}+\underbrace{f(n)}_{\text{计算过程中，除递归求解子问题以外，其它必要处理所花费的时间}}$$

对子问题递归求解花费的时间

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**：$T(n) = T(n/2)+1$

# 求解递归式

分治算法的计算时间表达式往往是递归式

**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Ο、Ω、Θ表示的函数式）。

## 求解递归式 预处理

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。

# 求解递归式 预处理

为便于处理，通常做如下假设和简化处理

## (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，可写为以下简单形式：

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

## 求解递归式 预处理

为便于处理，通常做如下假设和简化处理

**(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，可写为以下简单形式:

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


# 结论：



被简化的细节并不是不重要，只是对这些细节的处理不影响分析算法的渐近界，是在“无穷大”分析中做出的合理假设和简化。

在细节被简化处理的同时，也要知道它们在什么情况下是“实际”重要的。这样就可以了解算法在各种情况下的具体执行情况。

## 求解递归式 代换法

**先猜测解的形式，然后用数学归纳法验证猜测的正确性**

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（因此得名 “代换法” ），然后证明推论的正确性。

## 代换法 基本步骤

**(1) 猜测解的形式**

**(2) 用数学归纳法证明猜测的正确性**

## *Example：代换法*

例：用代换法确定下式的上界

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

该式与 $T(n) = 2T(n/2) + n$ 类似，故猜测其解为

T(n) = O(nlogn)

**代入法要证明的是：**如何恰当选择常数c，使得*T(n)≤cnlogn* 成立?

**所以现在设法证明：** *T(n)≤cnlogn*，并**确定常数c的存在**。

## Example：代换法

证明：

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

则在数学归纳法推论证明阶段对递归式做代换，有：

$$
\begin{aligned}
T(n) &\le 2(c\lfloor n/2 \rfloor log(\lfloor n/2 \rfloor)) + n \\
&\le cn\,log(n/2) + n \qquad \text{去掉底函数} \\
&= cn\,log\,n - cn\,log\,2 + n \qquad \text{对数运算} \\
&= \underline{cn\,log\,n - (c-1)n} \qquad \text{化简结果}
\end{aligned}
$$

故，要使**T(n)≤cnlogn**成立，只要c≥1就可以，这样的c是合理存在的。

## *Example：代换法*

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：$T(n) \le cn\log n$ **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：**$T(1) \le c \cdot 1 \cdot \log 1 = 0$**，**与T(1)=1不相符**。

## *Example：代换法*

**归纳证明的基础不成立，怎么处理?**

从$n_0$的性质出发：只需要存在常数$n_0$，使得$n \geq n_0$时结论成立即可，

所以$n_0$不一定取1。

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)

## Example： 代换法

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

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

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

  命题得证。

## 代换法 猜测递归式

### 并不存在通用的方法来猜测递归式的正确解

### 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 $T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n = O(n)$ 成立

但是错误，原因：并未证出一般形式T(n)≤cn成立（cn+n≰cn）

## 代换法 猜测递归式

并不存在通用的方法来猜测递归式的正确解

### 经验

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。

# Example：变量代换

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

分析：原始形态比较复杂

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

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

得：

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

## Example：变量代换

求：递归式 $T(2^m) \le 2T(2^{m/2}) + m$

再设 **S(m) = T(2^m)**，得以下形式递归式：$S(m) \le 2S(m/2) + m$

从而获得**形式上熟悉**的递归式。

根据前面的一些讨论，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，$T(n) = T(2^m)$

$$= S(m) = O(m\log m)$$

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

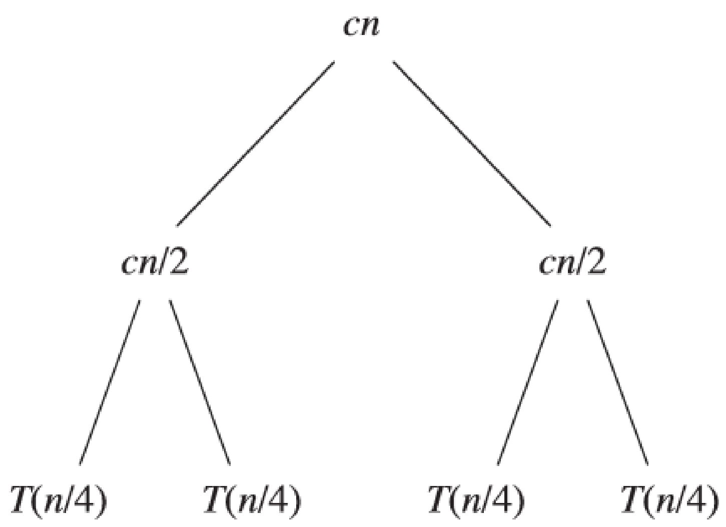
这里，$m = \log n$

# 求解递归式 递归树法

根据递归式的定义，可以画一棵**递归树：帮助我们猜测递归式的解。**

- 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

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



# 递归树法 时间分析

**节点代价：** 在递归树中，每个节点有求解相应（子）问题的代价(cost，这里指除递归以外的其它代价)。

**层代价：** 每一层各节点代价的和。

**总代价：** 整棵树的各层代价之和。

**目　标：** 利用树的性质，获取对递归式解的猜测，然后用代换法或其它方法加以验证。

# Example:递归树时间分析

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：为简单起见，对一些细节做必要、合理的简化和假设，这里为：

（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

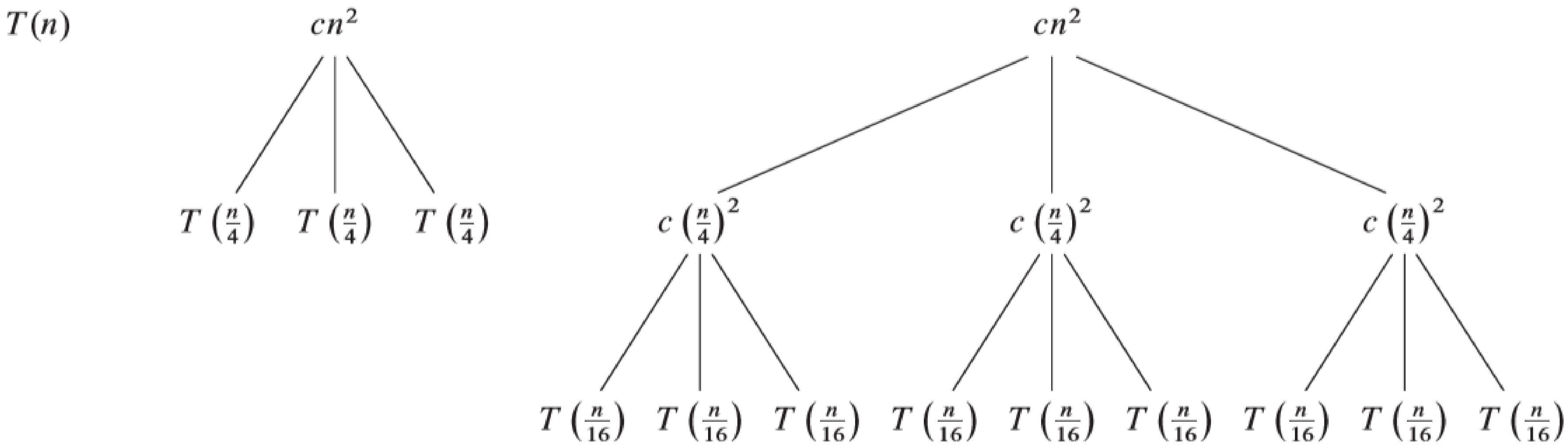
（3）**展开** $\Theta(n^2)$，代表递归式中非重要项。

- 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。

最终得到以下形式的递归式：$T(n)=3T(n/4)+cn^2$

# Example:递归树时间分析

例：已知递归式 $T(n)=3T(n/4)+cn^2$ ，求其上界




a) 对原始问题T(n)的描述。

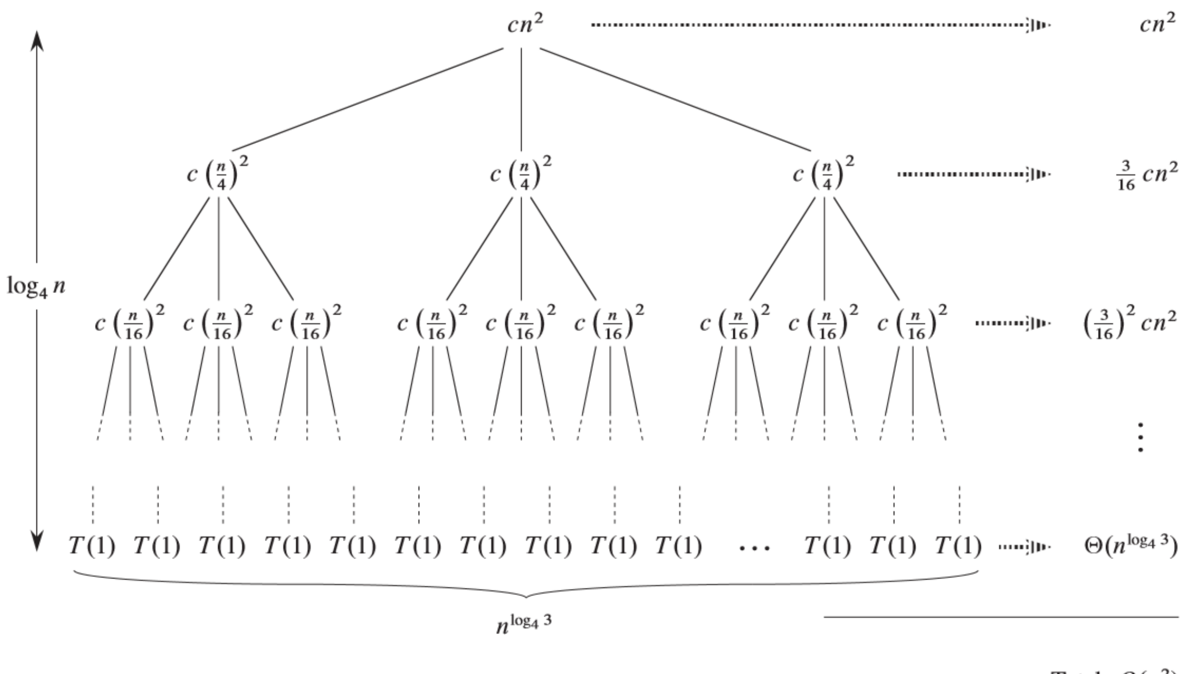
b) 第一层递归调用的分解情况，$cn^2$是顶层计算除递归以外的代价，T(n/4)是分解出来的规模为n/4的子问题的代价，总代价$T(n)=3T(n/4)+cn^2$。

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。

# Example:递归树时间分析

例：已知递归式 $T(n)=3T(n/4)+cn^2$ ，求其上界

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)

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)**

## Example:递归树时间分析

**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$3^i$。

# Example:递归树时间分析

## 代价计算

（1）**内部节点**：位于0~$\log_4 n - 1$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 $n^{\log_4 3}T(1) = \Theta(n^{\log_4 3})$

换底公式

$$\log_a^N = \frac{\log_m^N}{\log_m^a}$$

由换底公式可得：

$$a^{\log_b c} = c^{\log_b a}$$

# Example:递归树时间分析

## 树的总代价：各层代价之和

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

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

利用等比数列化简

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

- 对于实数x≠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 $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$
- 当和是无穷的且|x|<1时，得到无穷递减几何级数，此时 $\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$

## Example:递归树时间分析

T(n)中，cn²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(n²)**

# 代换法证明猜测的正确性

- 将T(n)≤dn²作为归纳假设，d是待确定的常数，带入推论证明过程，有

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

c是引入的另一个常量

显然，要使得T(n)≤dn²成立，只要d≥(16/13)c即可。所以，T(n)≤dn²的猜测成立。定理得证（边界条件的讨论略）。

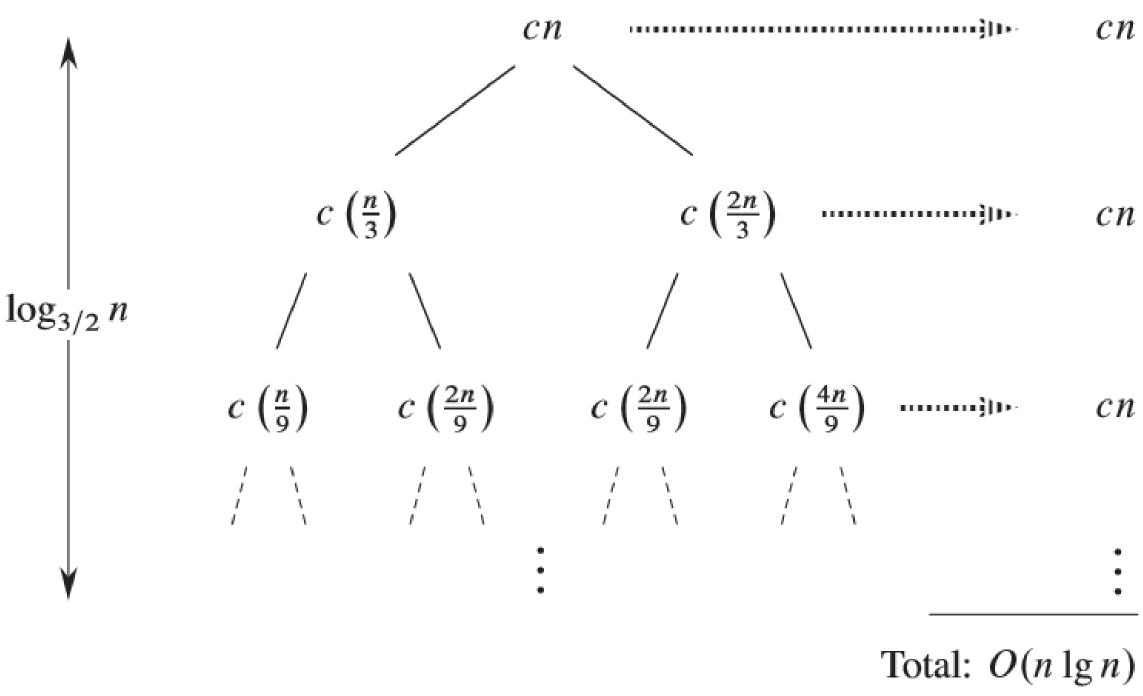
另： O(n²)是T(n)的一个紧确界，为什么?

## Example

例 求表达式 $T(n)=T(n/3)+T(2n/3)+O(n)$ 的上界

进一步地，**引入常数c**，展开O(n)，得：$T(n) \le T(n/3) + T(2n/3) + cn$

**递归树**为：



# Example

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

## Example

- **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：

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

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。

# Example

**猜测的证明**：证明O(nlogn)是递归式的上界

即证明**T(n)≤dnlog*n***，d是待确定的合适的正常数

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

成立吗?

**上式的成立条件**：$d \ge c/(\log 3 - (2/3))$，**存在!**

∴ 猜测正确，递归式解得证。

# 求解递归式 主方法

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≥1，b>1；f(n)是一个渐近正的函数。

**上式给出了算法总代价与子问题代价之间的关系，含义为：**

规模为n的原问题被分为a个子问题，每个子问题的规模是n/b。T(n)表示原始问题的时间，则每个子问题的时间为T(n/b)；问题分解及子问题解合并及其它有关运算的代价由函数f(n)描述。

# 主方法 主定理

设a≥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：

$T(n) = aT(n/b) + f(n)$　　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ε>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ε>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 $af(n/b) \le cf(n)$，则 $T(n) = \Theta(f(n))$ 。

## 主方法 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ε>0，f(n)必须渐近地小于 $n^{\log_b a}$，两者相差了一个 $n^{\varepsilon}$ 因子。

# 主方法 主定理

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 $af(n/b) \le cf(n)$。

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

- 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。

# Example：主定理

分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）

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ε=1，

所以对应主定理的第一种情况。

于是有：$T(n)=O(n^2)$

## Example：主定理

分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）

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)=Θ(log*n*)

## Example：主定理

分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）

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)=nlog*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ε≈0.2。同时，对足够大的n，

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

其中，**c = 3/4**。所以第三种情况成立，**T(n) = Θ(nlogn)**。

# Example：主定理

分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）

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)=nlog$n$ 渐进大于 $n^{\log_b a} = O(n)$

第三种情况成立吗?

$$n^x (\log n)^y < n^{x+\varepsilon}$$

## Example：主定理

分析递归式满足主定理的哪种情形，即可得到解（无需证明，保证正确）

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

f(n)=nlog$n$ 渐进大于 $n^{\log_b a}=n^{\log_2 2}=O(n)$

**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。**


# 思考：

華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

还有没有其它方法化简递归式？

# 求解递归式 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。

# Example：直接化简

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

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

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

$$\ldots$$

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

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

$$=3n/2-2$$

## Example：直接化简

例：化简递归式 $T(n)=2T(n/2)+n\log n$

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

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

# 最近点对问题

# 最近点对问题

## 问题描述

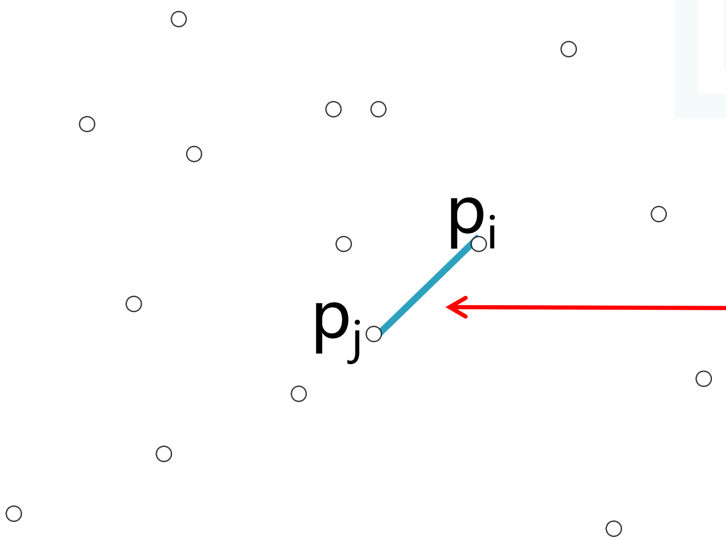
已知平面上分布着点集P中的n个点$p_1,p_2,\dots p_n$，点i的坐标记为$(x_i,y_i)$，$1\le i\le n$。两点之间的距离取其欧式距离，记为

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

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。

# 问题描述




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

平面上分布的点集

## 暴力搜索

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： $O(n^2)$，因为共计算 $n(n-1)/2$ 对点间距离。
- 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：$O(n^2)$。

## 分治求解

利用分治法“设计”一个具有O(nlogn)时间复杂度的算法求解

**1）首先将所有的点按照x坐标排序**

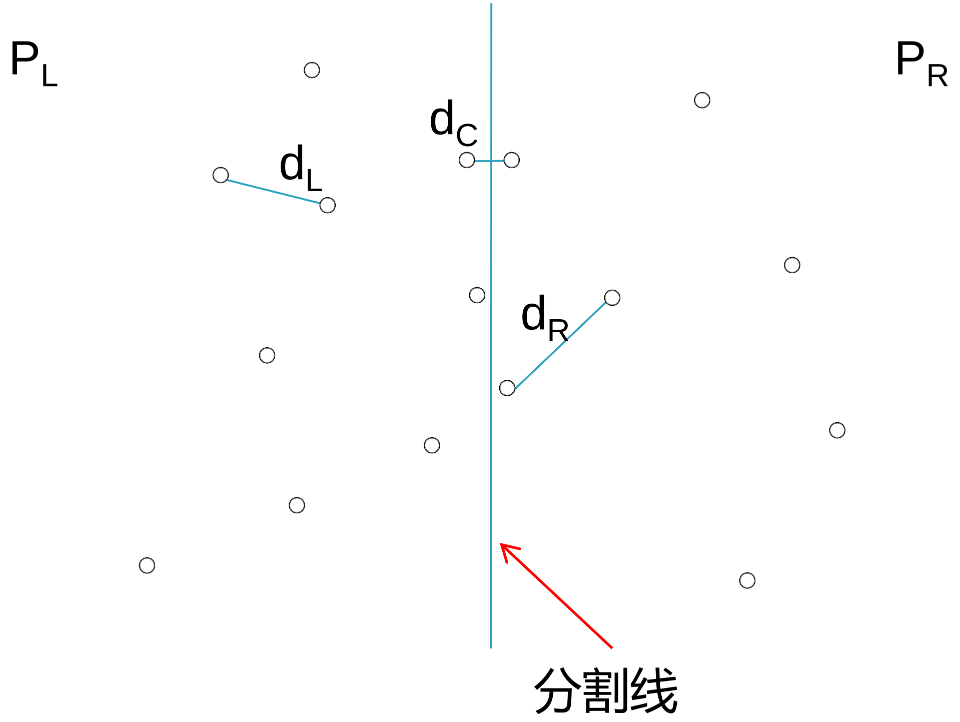
排序过程需要O($n\log n$)的时间，不会从整体上增加时间复杂度的数量级 (加法规则)。

**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$P_R$。

# 最近点对问题

## 分治求解




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）。

## 分治求解

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(nlogn)**以内。

# 分治求解

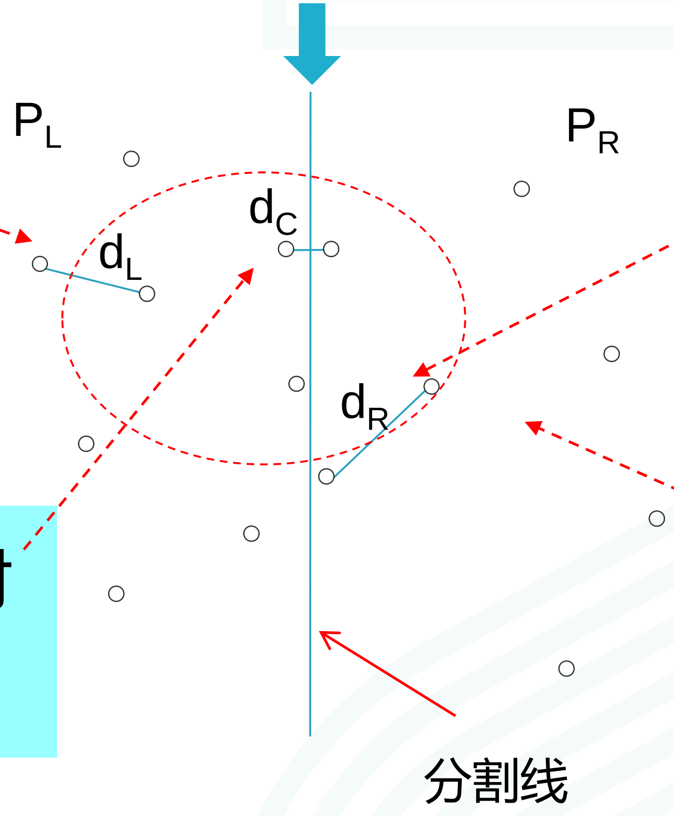
(1)做分割线（垂线），将点集分为$P_L$和$P_R$两部分

(2)递归地在$P_L$中的找具有$d_L$的点对

(3)递归地在$P_R$中的找具有$d_R$的点对

(4)在跨越分割线的点对中找具有$d_C$的点对

(5)返回$\min(d_L,d_R,d_C)$对应的点对

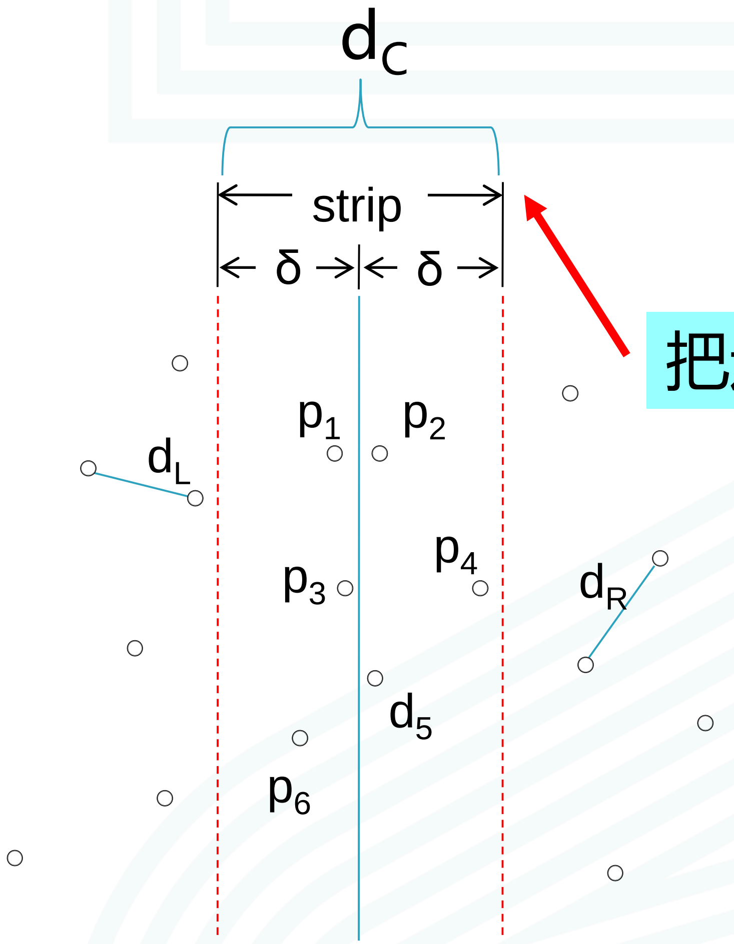


# 分治求解

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

从而，对$d_C$的搜索只限制在strip区域内，这样就限定了需要考虑的点的个数。

超出带区域的$d_c$是无用的




把这个区域叫做带(strip)

## 分治求解 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* ← $O(\sqrt{n})$

*if dist($p_i$,$p_j$)<δ*

*δ = dist($p_i$,$p_j$);* ← δ在不断的修正中

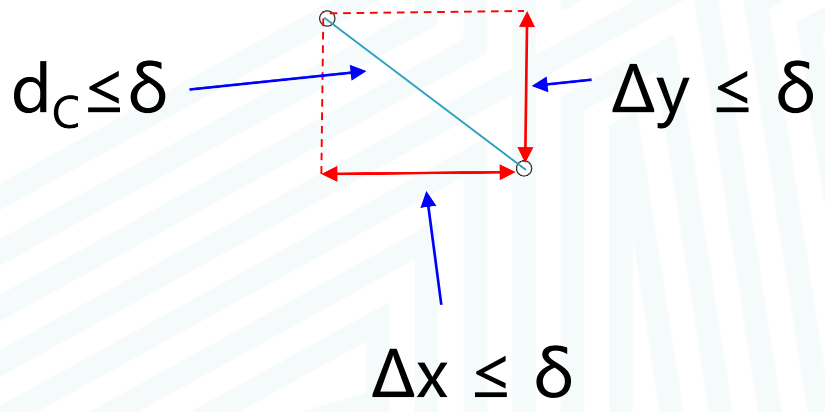
# 分治求解 计算$d_C$

## 方法一可能存在的问题

- 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。

### 改进方案

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$d_C$>δ。



## 分治求解 计算$d_C$

**改进方案** 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$p_i$而言更远的结点就没必要搜索了，转而处理$p_i$后面的点$p_{i+1}$。

*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*

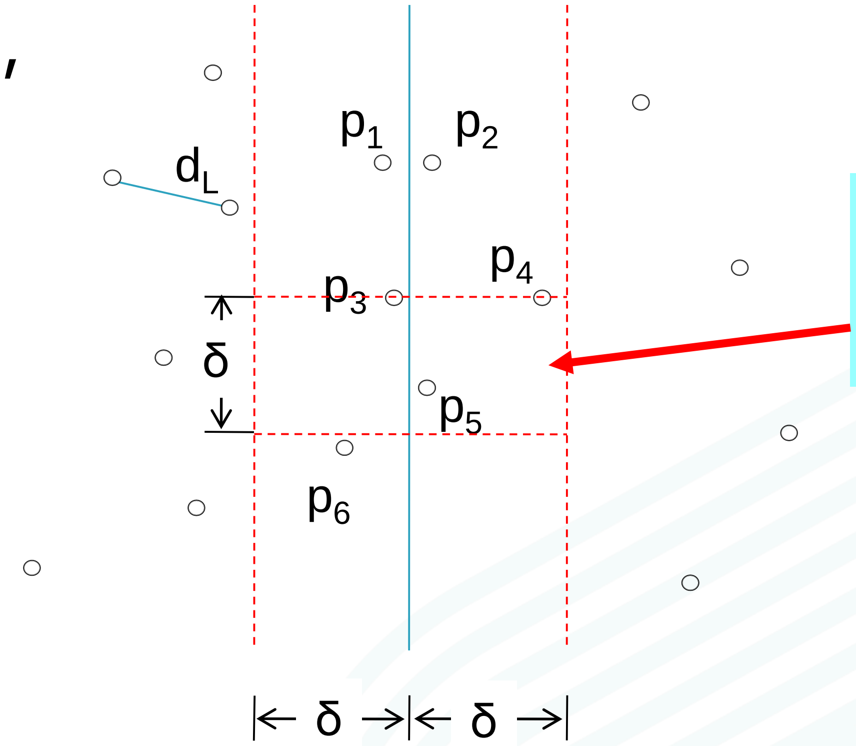
*break;*

*else if dist($p_i$,$p_j$)<δ* ← $\Delta y < \delta$

*δ = dist($p_i$,$p_j$);*

## 分治求解 计算$d_C$

改进方案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，

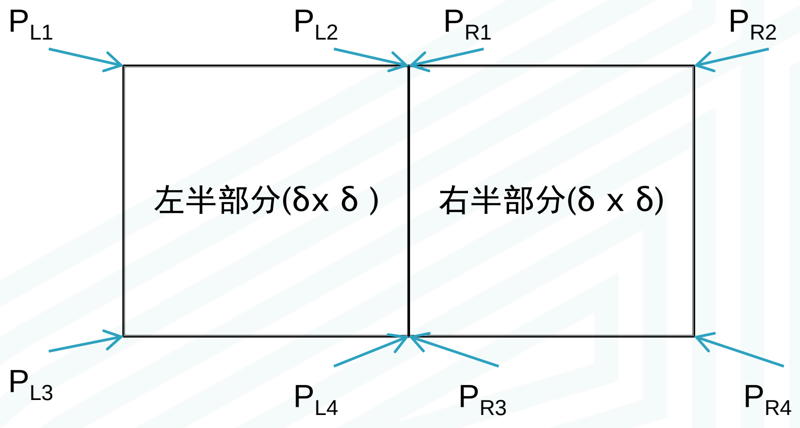



对于$p_3$只有两个点$p_4$和$p_5$落在垂直距离δ之内的带状区域中。

## 分治求解 计算$d_C$

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。

- 这是因为，最坏情况下，$p_i$和$p_j$点必定落在该带状区域左半部分的δXδ方块内或者右半部分的δXδ方块内。
- 每个方块最多包含4个点，且分别落在四个角上，并且其中一个是$p_i$，另外7个就是需要考虑的$p_j$点。如图所示：



## 分治求解 计算$d_C$

这样，对于每个$p_i$，最多有7个$p_j$要考虑，也就是是**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)，即使在最坏的情况下。

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：

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

**那么最近点对问题就可以得到O(nlogn)的解了吗？**

## 分治求解 计算$d_C$

还有问题需要讨论：点y坐标的排序问题

**问题所在：**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(nlogn)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?**

## 分治求解 计算$d_C$

**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(nlogn)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。

## 分治求解 计算$d_C$

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**。此时Q中就只含有带中的点，而且这些点已是按照y坐标排好序了的。这一处理需要O(n)的时间。下一步，对每个$p_i$，寻找近邻中$\Delta y \le \delta$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

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

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

## 问题扩展

坐标（x1, y1）的点P1与坐标（x2, y2）的点P2的曼哈顿距离为

$$|x1 - x2| + |y1 - y2|.$$


在国际象棋棋盘上，有这种横平竖直的格子，描述格子和格子之间的距离可以直接用曼哈顿距离。如 A1 格子到 C4 格子的曼哈顿距离计算如下

$$c = |3-1| + |4-1| = 5$$

## 问题扩展

坐标（x1, y1）的点P1与坐标（x2, y2）的点P2的曼哈顿距离为

$$|x1 - x2| + |y1 - y2|.$$

# 问题扩展

距离是一门关于范数的学问

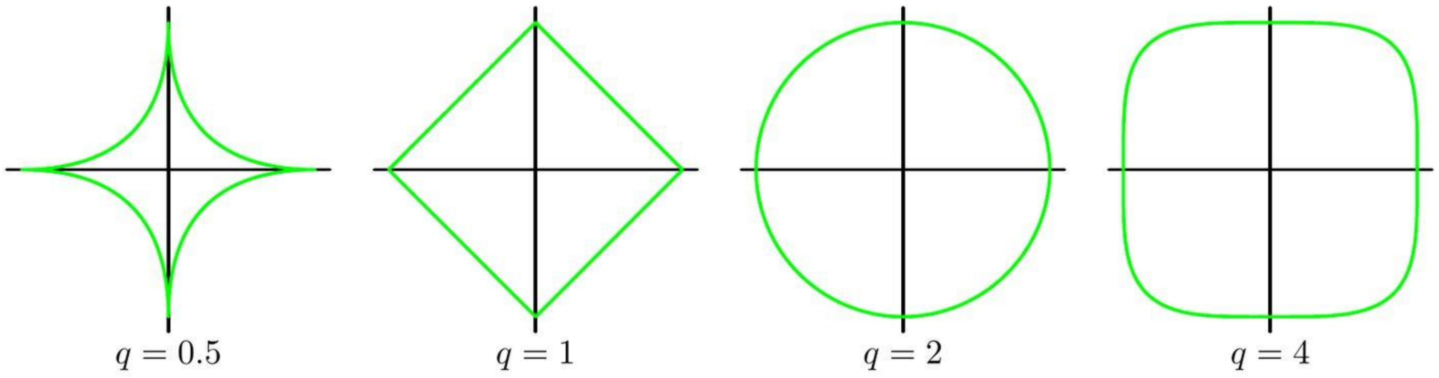



**Figure 3.3** Contours of the regularization term in (3.29) for various values of the parameter $q$.

$$||x||_1 = \sum_{i=1}^{N} |x_i| \qquad ||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{N} x_i^2}$$

$$||\mathbf{x}||_{-\infty} = \min_i |x_i| \qquad ||\mathbf{x}||_p = (\sum_{i=1}^{N} |x_i|^p)^{\frac{1}{p}} \qquad ||\mathbf{x}||_\infty = \max_i |x_i|$$