# 程序设计方法与实践

——分治法

# 第五章 分治法

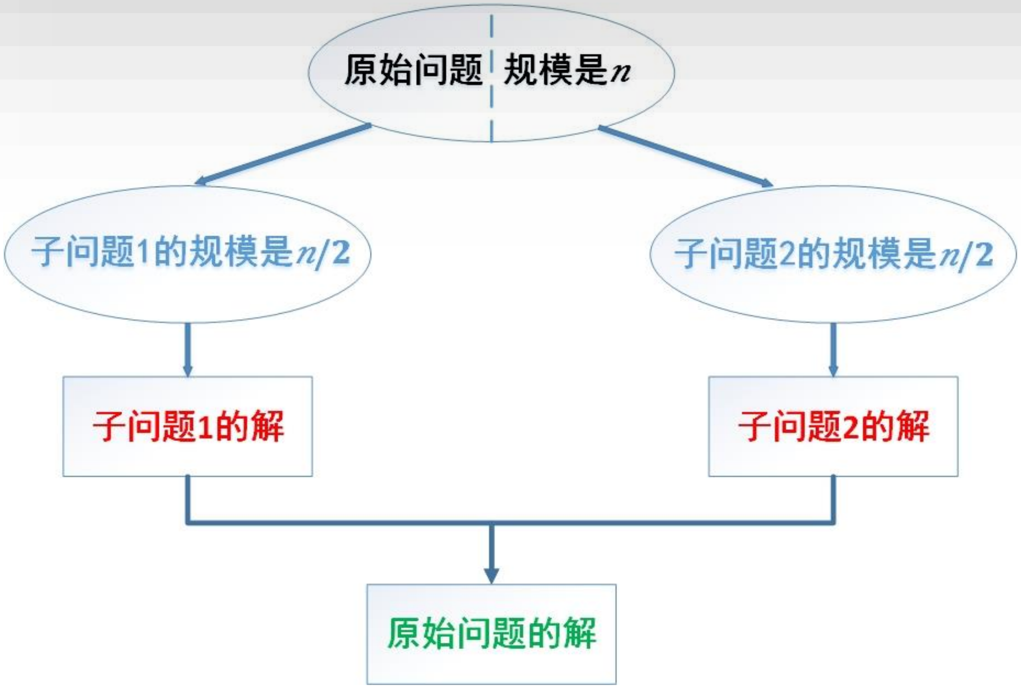
- 分治法的定义
- 主定理与分治法
- 合并排序
- 大整数乘法
- 快速排序及矩阵乘法

# 分治法：分而治之

1. 将原始问题划分为若干同类型子问题，子问题最好规模相同；
2. 对子问题求解（通常使用递归方法）；
3. 将子问题的求解结果合并，得到原问题的解。

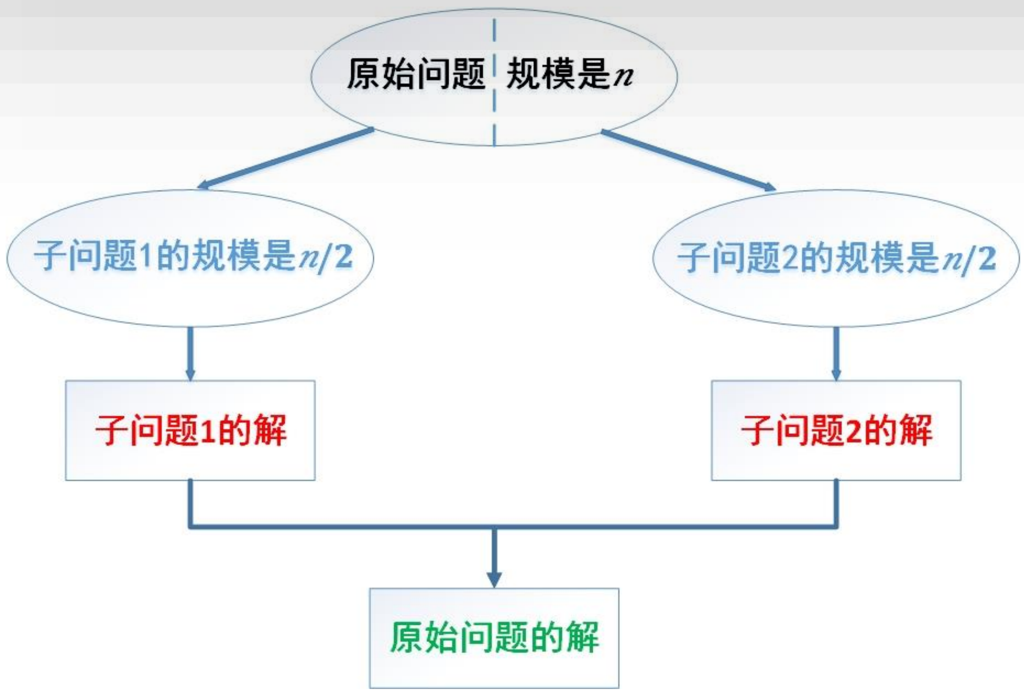
# 分治法：分而治之

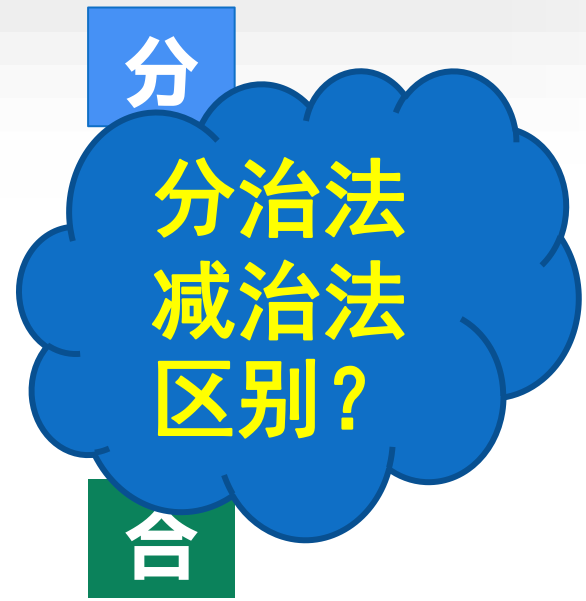
1. 将原始问题划分为若干同类型子问题，子问题最好规模相同；
2. 对子问题求解（通常使用递归方法）；
3. 将子问题的求解结果合并，得到原问题的解。



# 分治法：分而治之

1. 将原始问题划分为若干同类型子问题，子问题最好规模相同；
2. 对子问题求解（通常使用递归方法）；
3. 将子问题的求解结果合并，得到原问题的解。



# 主定理与分治法

## 主定理

- **对常数**$a > 0$、$b > 1$**及**$d \geq 0$，**有**$T(n) = aT(\lceil n/b \rceil) + O(n^d)$**成立，则，**

$$T(n) = \begin{cases} O(n^d), & \text{如果} d > \log_b a \\ O(n^d logn), & \text{如果} d = \log_b a \\ O(n^{\log_b a}), & \text{如果} d < \log_b a \end{cases}$$

- 1、$f(n) \in O(n^{\log_b a-\varepsilon})$ $\quad \varepsilon > 0$

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

- 2、$f(n) \in \Theta(n^{\log_b a} \log^k n)$ $\quad k \geq 0$

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

- 3、$f(n) \in \Omega(n^{\log_b a+\varepsilon})$ $\quad \varepsilon > 0$

$$T(n) = \Theta(f(n))$$

# 主定理与分治法

## 主定理

- **对常数**$a > 0$、$b > 1$**及**$d \geq 0$，**有**$T(n) = aT(\lceil n/b \rceil) + O(n^d)$**成立，则**，

$$\boldsymbol{T(n)} = \begin{cases} O(n^d), & \text{如果} d > \log_b a \\ O(n^d logn), & \text{如果} d = \log_b a \\ O(n^{\log_b a}), & \text{如果} d < \log_b a \end{cases}$$

分治算法遵循一种通用模式，即：在解决规模为$n$的问题时，总是先递归地分解$\boldsymbol{a}$**个**规模为$\boldsymbol{n/b}$**的子问题**，然后在$O(n^d)$时间内将子问题的解合并，其中$a, b, d > 0$是一些特定的正数。

# 合并排序（归并排序）

**合并排序**：需要排序的数组$A[0, \ldots n-1]$

- 一分为二：$A[0, \ldots \lfloor n/2 \rfloor]$和$A[\lfloor n/2 \rfloor, \ldots n-1]$
- 对每个子数组以相同方式递归排序
- 将排好序的子数组合并为一个有序数组。

- **合并排序算法**

**算法** Mergesort($A[0, \ldots, n-1]$)

//递归调用mergesort来对数组$A[0, \ldots, n-1]$排序

//输入：一个可排序数组$A[0, \ldots, n-1]$

//输出：非降序排列的数组$A[0, \ldots, n-1]$

**if** $n > 1$

copy $A[0, \ldots, \lfloor n/2 \rfloor - 1]$ **to** $B[0, \ldots, \lfloor n/2 \rfloor - 1]$

copy $A[\lfloor n/2 \rfloor, \ldots, n-1]$ **to** $C[0, \ldots, \lceil n/2 \rceil - 1]$

Mergesort($B[0, \ldots, \lfloor n/2 \rfloor - 1]$)

Mergesort($C[0, \ldots, \lceil n/2 \rceil - 1]$)

Merge(B,C,A) //合并算法

# 合并排序（归并排序）

**合并排序**：需要排序的数组$A[0,\dots n-1]$

- 一分为二：$A[0,\dots\lfloor n/2\rfloor]$和$A[\lfloor n/2\rfloor,\dots n-1]$
- 对每个子数组以相同方式递归排序
- 将排好序的子数组合并为一个有序数组。

- **合并算法**

**算法 Merge**$(B[0,\dots,p-1], C[0,\dots,q-1], A[0,\dots,p+q-1])$

//将两个有序数组合并为一个有序数组

//输入：两个有序数组$B[0,\dots,p-1], C[0,\dots,q-1]$

//输出：$A[0,\dots,p+q-1]$中已经有序存放了$B$和$C$的元素

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

**while** $i < p$ **and** $j < q$ **do**

  **if** $B[i] \le C[j]$  $A[k] \leftarrow B[i];\ i \leftarrow i+1;$

  **else** $A[k] \leftarrow C[j];\ j \leftarrow j+1;$

  $k \leftarrow k+1;$

**if** $i = p$ **copy** $C[j..q-1]$ **to** $A[k..p+q-1]$

**else** **copy** $B[i..p-1]$ **to** $A[k..p+q-1]$

# 合并排序（归并排序）

- **合并排序算法**

**算法** Mergesort$(A[0,\ldots,n-1])$

//递归调用mergesort来对数组$A[0,\ldots,n-1]$排序
//输入：一个可排序数组$A[0,\ldots,n-1]$
//输出：非降序排列的数组$A[0,\ldots,n-1]$
**if** $n>1$
  copy $A[0,\ldots,\lfloor n/2\rfloor-1]$ **to** $B[0,\ldots,\lfloor n/2\rfloor-1]$
  copy $A[\lfloor n/2\rfloor,\ldots,n-1]$ **to** $C[0,\ldots,\lceil n/2\rceil-1]$
  Mergesort$(B[0,\ldots,\lfloor n/2\rfloor-1])$
  Mergesort$(C[0,\ldots,\lceil n/2\rceil-1])$
  Merge(B,C,A) //合并算法

- **合并算法**

**算法** **Merge**$(B[0,\ldots,p-1],C[0,\ldots,q-1],A[0,\ldots,p+q-1])$

//将两个有序数组合并为一个有序数组
//输入：两个有序数组$B[0,\ldots,p-1],C[0,\ldots,q-1]$
//输出：$A[0,\ldots,p+q-1]$中已经有序存放了$B$和$C$的元素
$i\leftarrow 0; j\leftarrow 0; k\leftarrow 0$
**while** $i<p$ **and** $j<q$ **do**
  **if** $B[i]\leq C[j]$  $A[k]\leftarrow B[i];\ i\leftarrow i+1;$
  **else** $A[k]\leftarrow C[j];\ j\leftarrow j+1;$
  $k\leftarrow k+1;$
**if** $i=p$ **copy** $C[j..q-1]$ **to** $A[k..p+q-1]$
**else** **copy** $B[i..p-1]$ **to** $A[k..p+q-1]$

- copy操作需要线性时间
- Merge操作需要线性时间
- 基本操作：Mergesort$(n)$，需要需要$T(n)$时间

# 合并排序（归并排序）

## 分治法之合并排序

- **分**：把$A$数组一分为二：$B = A[0, \dots \lfloor n/2 \rfloor]$和$C = A[\lfloor n/2 \rfloor, \dots n-1]$.
- **治**：求解Mergesort($B$)和Mergesort($C$).
- **合**：在$O(n)$时间内计算Merge(B,C,A).
- **递推式**：$\boldsymbol{T(n) = 2T(n/2) + O(n)}$

### 主定理

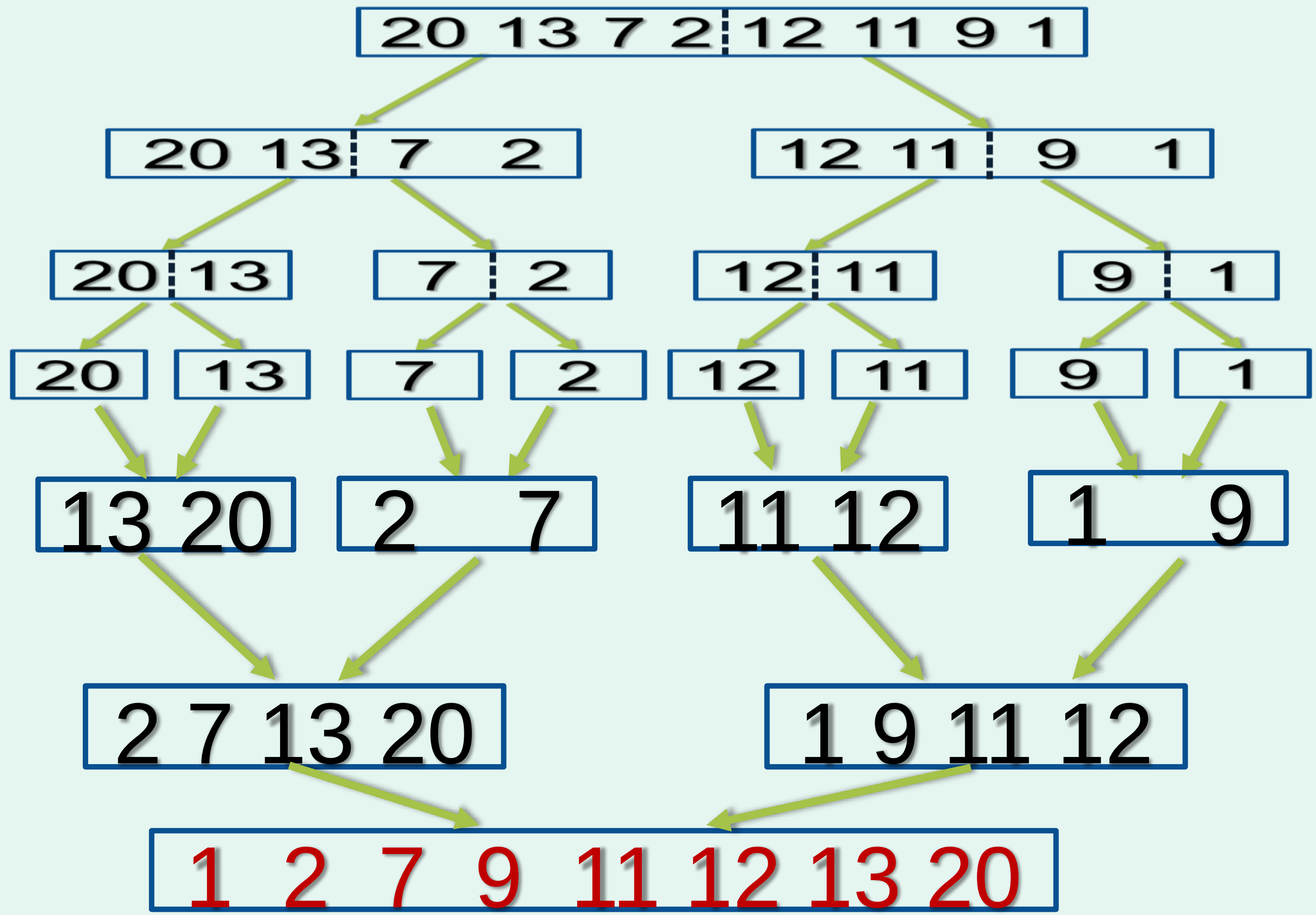
- 对常数$\boldsymbol{a} > \boldsymbol{0}$、$b > \boldsymbol{1}$及$d \geq 0$，有$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$成立，则，

$$\boldsymbol{T(n)} = \begin{cases} O(n^d), & \text{如果} d > \log_b a \\ O(n^d logn), & \text{如果} d = \log_b a \\ O(n^{\log_b a}), & \text{如果} d < \log_b a \end{cases}$$

因为$a = 2, b = 2, d = 1,$
$log_b a = 1, d = 1,$ 所以，
$$\boldsymbol{T(n) = O(nlogn)}$$

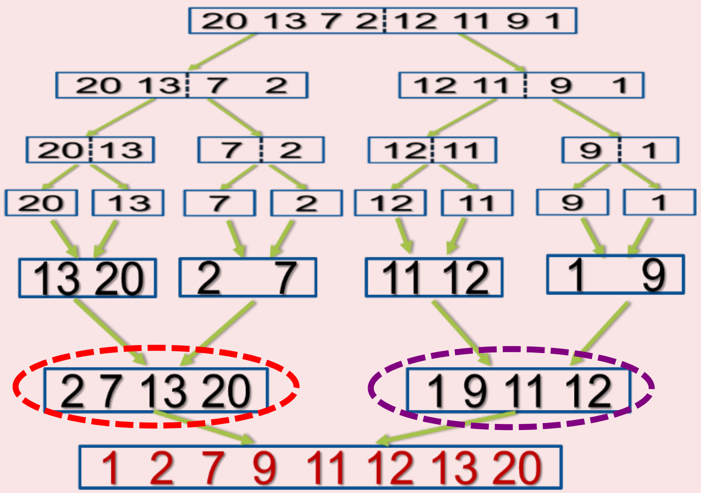
# 合并排序（归并排序）

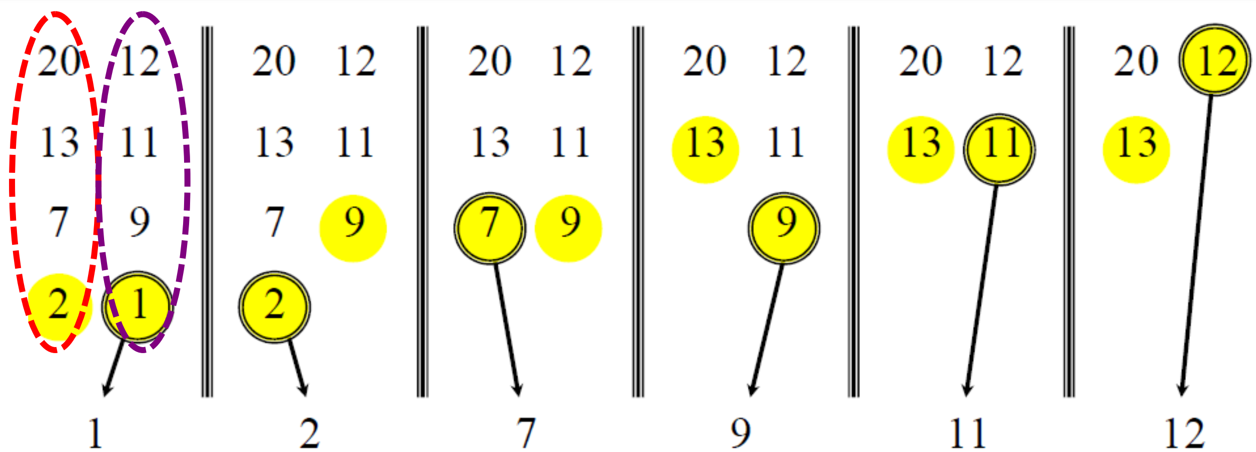
**示例**：演示对数列$A[0,\ldots 7]=\{20,13,7,2,12,11,9,1\}$合并排序的过程。



# 合并排序（归并排序）

**示例**：演示对数列$A[0,\ldots 7]=\{20,13,7,2,12,11,9,1\}$合并排序的过程。



# 大整数乘法

**大整数乘法，如对>100位的十进制/二进制整数做乘法。**

- **常规整数相乘**，如，$n(n=4)$位数1536和2487，计算如下：

$$1536 = 15 \times 10^2 + 36 = 1 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$$

$$2487 = 24 \times 10^2 + 87 = 2 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$$

$$\therefore \quad 1536 \times 2487 = (1 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 6 \times 10^0) \times (2 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 7 \times 10^0)$$

共使用$n^2 = \mathbf{4} \times \mathbf{4} = \mathbf{16}$**次位乘**。

# 大整数乘法

**大整数乘法，如对>100位的十进制/二进制整数做乘法。**

**二进制大整数乘法**，长度都为$n$的二进制数$x, y$的乘法运算，

$$x = [\text{左半部分}][\text{右半部分}] = [x_L][x_R]$$

$$y = [\text{左半部分}][\text{右半部分}] = [y_L][y_R]$$

$$x = [x_L][x_R] = 2^{n/2}x_L + x_R$$

$$y = [y_L][y_R] = 2^{n/2}y_L + y_R$$

$$x \times y = \left(2^{n/2}x_L + x_R\right)\left(2^{n/2}y_L + y_R\right)$$

$$= 2^n x_L y_L + 2^{n/2}\left(\boxed{x_L y_R + x_R y_L}\right) + x_R y_R$$

$$\boxed{x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R}$$

$$x \times y = 2^n x_L y_L + 2^{n/2}\left((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\right) + x_R y_R$$

$$x \times y = (2^n - 2^{n/2})x_L y_L + 2^{n/2}(x_L + x_R)(y_L + y_R) + (1 - 2^{n/2})x_R y_R$$

# 大整数乘法

**二进制大整数乘法**，长度都为$\boldsymbol{n}$的二进制数$\boldsymbol{x}, \boldsymbol{y}$的乘法运算，

$$x \times y = (2^n - 2^{n/2})x_L y_L + 2^{n/2}(x_L + x_R)(y_L + y_R) + (1 - 2^{n/2})x_R y_R$$

- 加法操作需要线性时间
- 乘以2的幂次方的操作需要线性时间（相当于移位）
- 基本操作：$x_L y_L, (x_L + x_R)(y_L + y_R), x_R y_R$

$x_L y_L, (x_L + x_R)(y_L + y_R), x_R y_R$**是什么?**

长度为$n/2$位的**二进制大整数乘法!**

**二进制大整数乘法**，长度为$\boldsymbol{n}$的二进制数$\boldsymbol{x}, \boldsymbol{y}$的乘法运算，

1. 调用3个$n/2$位二进制乘法子问题，
2. 在$O(n)$时间内汇总计算最终结果。

**分治法!**

# 大整数乘法

## 分治法之大整数乘法

- **分**：长度为$n$的二进制数$x, y$划分成左右部分$x_L, x_R, y_L, y_R$.
- **治**：求解$x_L y_L, (x_L + x_R)(y_L + y_R), x_R y_R$
- **合**：在$O(n)$时间内**加法**计算最终结果：

$$2^n x_L y_L + 2^{n/2}\left((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\right) + x_R y_R$$

- **递推式**：$\boldsymbol{T(n) = 3T(n/2) + O(n)}$

### 主定理

- 对常数$a > 0$、$b > 1$及$d \geq 0$，有$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$成立，则，

$$T(n) = \begin{cases} O(n^d), & 如果 d > \log_b a \\ O(n^d \log n), & 如果 d = \log_b a \\ O(n^{\log_b a}), & 如果 d < \log_b a \end{cases}$$

因为$a = 3, b = 2, d = 1$,
$\log_b a \approx 1.59, d < 1.59$ 所以,

$$\boldsymbol{T(n) = O(n^{1.59})}$$

常规乘法

**位乘次数是位数乘积：$n^2$**

# 快速排序

算法思路:

对于输入$A[0..n-1]$，按以下三个步骤进行排序:

(1)**<u>分区</u>**:取$A$中的一个元素为中心点(pivot) 将$A[0..n-1]$划分成3段: $A[0..s-1]$,$A[s]$,$A[s+1..n-1]$,使得

$A[0..s-1]$中任一元素$\leq A[s]$,

$A[s+1..n-1]$中任一元素$\geq A[s]$;

下标$s$在划分过程中确定。

(2)**<u>递归求解</u>**:递归调用快速排序法分别对$A[0..s-1]$和$A[s+1..n-1]$排序。

(3)**<u>合并</u>**:合并A[0..s-1], A[s], A[s+1..n-1]为A[0..n-1]

# 快速排序

**快速排序算法 QuickSort(A[l..r])**

**// 使用快速排序法对序列或者子序列排序**

**// 输入：子序列A[l..r]或者序列本身A[0..n-1]**

**// 输出：非递减序列A**

**if l < r**

**s ← Partition( A[l..r] )**

**QuickSort( A[l..s-1] )**

**QuickSort( A[s+1..r] )**

**//s是中轴元素/基准点，是数组分区位置的标志**

**中轴元素如何选?**

# 快速排序

## Partition划分算法:

- 1)上节课讲过的的Lomuto算法
- 2)霍尔两次扫描法。

**算法** LomutoPartition($A[l..r]$)

//采用 Lomuto 算法，用第一个元素作为中轴对子数组进行划分

//输入：数组 $A[0..n-1]$ 的一个子数组 $A[l..r]$，它由左右两边的索引 $l$ 和 $r$ ($l \leqslant r$)定义

//输出：$A[l..r]$的划分和中轴的新位置

$p \leftarrow A[l]$

$s \leftarrow l$

**for** $i \leftarrow l+1$ **to** $r$ **do**

  **if** $A[i] < p$

    $s \leftarrow s+1$; swap($A[s]$, $A[i]$)

swap($A[l]$, $A[s]$)

**return** $s$

若第一个元素较大
则交换次数多

# 分区算法

**快排算法使用了霍尔(A.R. Hoare)两次扫描方法:**

与Lomuto算法不同，从子数组的两端扫描与中轴元素比较。

- 指针$i$从数组左边开始扫描，忽略小于中轴的元素，遇到大于等于中轴的元素$A[i]$时停止。
- 指针$j$从数组右边开始扫描，忽略大于中轴的元素，遇到小于等于中轴的元素$A[j]$时停止，然后交换$A[i]$和$A[j]$。
- 指针不相交则继续。

# 分区算法

两次扫描全部停止以后，取决于扫描的指针是否相交，会发生 3 种不同的情况。如果扫描指针 $i$ 和 $j$ 不相交，也就是说 $i<j$，我们简单地交换 $A[i]$和 $A[j]$，再分别对 $i$ 加 1，对 $j$ 减 1，然后继续开始扫描。

| | | $\rightarrow i$ | | $j \leftarrow$ | |
|---|---|---|---|---|---|
| $p$ | 全部$\le p$ | $\ge p$ | … | $\le p$ | 全部$\ge p$ |

如果扫描指针相交，也就是说 $i>j$，把中轴和 $A[j]$交换以后，我们得到了该数组的一个划分。

| | | $j \leftarrow$ | $\rightarrow i$ | |
|---|---|---|---|---|
| $p$ | 全部$\le p$ | $\le p$ | $\ge p$ | 全部$\ge p$ |

最后，如果扫描指针停下来时指向的是同一个元素，也就是说 $i=j$，被指向元素的值一定等于 $p$（为什么？）。因此，我们建立了该数组的一个分区：

| | | $\rightarrow i=j \leftarrow$ | |
|---|---|---|---|
| $p$ | 全部$\le p$ | $=p$ | 全部$\ge p$ |

# 数组的分区算法

**算法 HoarePartition($A[l..r]$)**

**//以第一个元素为中轴，对子数组进行划分**

**//输入：数组$A[0 \ldots n-1]$的子数组$A[l..r]$**

**//输出：$A[l \ldots r]$的一个划分，分裂点的位置作为函数的返回值**

$p \leftarrow A[l]$

$i \leftarrow l$; $j \leftarrow r+1$; //注意i,j初始值是在有效范围之外

**repeat**

  **repeat** $i \leftarrow i+1$ **until** $A[i] \geq p$ //先改变指针再判断

  **repeat** $j \leftarrow j-1$ **until** $A[j] \leq p$

  $swap(A[i], A[j])$ //先交换后比较可能多造成一次交换

**until** $i \geq j$

**swap($A[i]$, $A[j]$)** // 当$i \geq j$，撤销最后一次交换

**swap($A[l]$, $A[j]$)** **// 把中轴的值放到对应位置**

**return** $j$;

# 快速排序的例子（双向扫描）

例如：$n = 8$

初始数组 $A[0..n-1] = [8, 4, 1, 7, 11, 5, 6, 9]$,

取元素$A[0] = 8$作为分裂点,

位置： 0 1 2 3 4 5 6 7

1. {8, 4, 1, 7,11, 5, 6, 9}

$i\uparrow$ $j\uparrow$ 指针$i$、$j$分别向中间移动

2. {8, 4, 1, 7,11, 5, 6, 9}

$i\uparrow$ $j\uparrow$ 符合条件，指针停止

3. {8, 4, 1, 7, 6, 5, 11, 9}

$i\uparrow$ $j\uparrow$ 数据交换,

# 快速排序的例子（双向扫描）

例如：$n = 8$

初始数组 $A[0..n-1] = [8, 4, 1, 7, 11, 5, 6, 9]$,

取元素$A[0] = 8$作为分裂点,

位置： 0 1 2 3 4 5 6 7

4. {8, 4, 1, 7, 6, 5, 11, 9}

j↑ $i$↑ i,j继续移动

5. {8, 4, 1, 7, 6, 11, 5, 9}

j↑ $i$↑ 数据交换，满足i<j跳出循环

6. {8, 4, 1, 7, 6, 5, 11, 9}

j↑ $i$↑ 撤销最后一次交换

7. {5, 4, 1, 7, 6, 8,11, 9} 中轴值放入对应位置

# 快速排序的例子（双向扫描）

分解得：

A[0..s-1]=[5, 4, 1, 7, 6]; A[s]=8; A[s+1..7]=[11,9]; s=5

排序:

A[0..s-1]=[1, 4, 5, 6, 7]; A[ s+1..n-1]=[9, 11]。

合并:

把A[0..s-1]中的元素放在分裂点元素8之前, A[s+1..n-1]中的元素放在分裂点元素之后, 结果[1, 4, 5, 6, 7, **8**, 9, 11]

# 快速排序效率分析

基本操作：**比较**（划分算法中比较n次）

最优情况下：所有分裂点均处中部

当$n > 1$时，$C_{best}(n) = 2C_{best}(n/2) + n$

$$C_{best}(1) = 0$$

由主定理解得 $C_{best}(n) \in \Theta(nlog_2 n)$

# 快速排序效率分析

最坏情况下：所有分裂点均处于极端

在进行 $n+1$ 次比较( $i$，$j$ 指针交叉)后建立了分区，还会对数组进行排序，继续到最后一个子数组 $A[n-2..n-1]$。总比较次数为：

$$\begin{aligned} C(n) &= (n+1)+n+\cdots+3 \\ &= (n+2)(n+1)/2-3 \\ &\in \Theta(n^2) \end{aligned}$$

最坏时间复杂度：$O(n^2)$
平均时间复杂度：$O(nlogn)$
稳定性：不稳定

# 快速排序不稳定的例子

{6, 7, 5, 2, $\overline{5}$, 8}　初始序列

{6, $\overline{5}$, 5, 2, 7, 8}

{2, $\overline{5}$, 5, 6, 7, 8}　结果序列

**快速排序具有不稳定性。**

# 二叉树遍历及其相关特性

所谓二叉树的遍历指的是遵循某一种次序来访问二叉树上的所有结点，使得树中每一个结点被访问了一次且只访问一次。

由于二叉树是一种非线性结构，树中的结点可能有不止一个的直接后继结点，所以遍历前必须先规定访问的次序。

# 中序遍历（INORDER TRAVERSAL）

**二叉树的中序遍历算法比较简单，使用递归的策略。**

**在遍历以前首先确定遍历的树是否为空，如果为空，则直接返回；否则中序遍历的算法步骤如下：**

**（1）对左子树L执行中序遍历算法**

**（2）访问输出根结点V的值。**

**（3）对右子树R执行中序遍历算法。**

# 前序遍历（PREORDER TRAVERSAL）

**有了上面的中序遍历的过程，前序遍历也是类似的。**

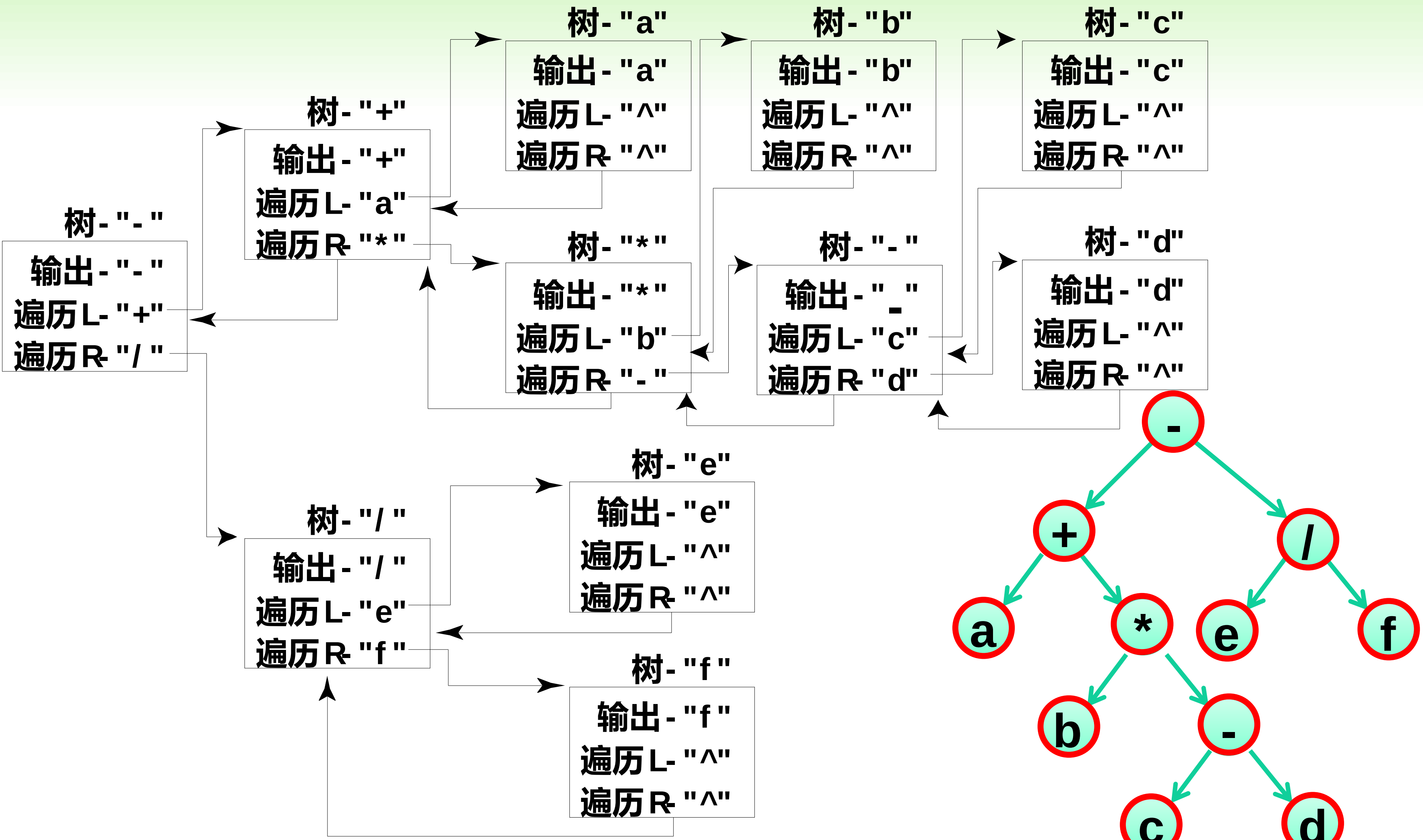
**在遍历以前首先确定遍历的树是否为空，如果为空，则直接返回；否则前序遍历的算法步骤如下：**

**（1）访问输出根结点V的值；**

**（2）对左子树L执行前序遍历算法。**

**（3）对右子树R执行前序遍历算法。**

# 前序遍历执行过程图




– + a * b – c d / e f

# Strassen矩阵乘法

矩阵乘法是线性代数中最常见的运算之一，它在数值计算中有广泛的应用。若A和B是2个$n \times n$的矩阵，则它们的乘积$C = A \times B$同样是一个$n \times n$的矩阵。A和B的乘积矩阵C中的元素$c[i,j]$定义为:

$$\mathrm{C}[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j]$$

若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素$c[i,j]$，需要做$n$个乘法和$n-1$次加法。因此，求出矩阵C的$n^2$个元素所需的计算时间为$O(n^3)$。

# Strassen矩阵乘法

- Strassen采用了分治技术，将计算2个$n$阶矩阵乘积所需的计算时间改进到

$$O(n^{log_2 7}) = O(n^{2.807})。$$

- 首先，假设$n = 2^k$。将矩阵A，B和C中每一矩阵都分块成为4个大小相等的子矩阵，每个子矩阵都是$n/2 \times n/2$的方阵。由此可将方程$C = A \times B$重写为:

# Strassen矩阵乘法

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

**其中：**

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

# Strassen矩阵乘法

则2个2阶方阵的乘积可以直接用上式计算出来，共需8次乘法和4次加法。当子矩阵的阶大于2时，为求2个子矩阵的积，可以继续将子矩阵分块，直到子矩阵的阶降为2。依此算法，计算2个n阶方阵的乘积转化为计算8个$n/2$阶方阵的乘积和4个$n/2$阶方阵的加法（可在$n^2$内完成）。因此，上述分治法的计算时间耗费$T(n)$应该满足：

$$\begin{cases} T(n) = 8T(n/2) + n^2 \quad n > \\ = 2 \end{cases}$$

$$T(1) = 1$$

$\mathrm{d} = 2 < 3,$
使用**主方法1**

这个递归方程的解仍然是$T(n) = O(n^3)$

# Strassen矩阵乘法

Strassen提出了一种新的算法来计算2个2阶方阵的乘积。他的算法只用了7次乘法运算，但增加了加、减法的运算次数。这7次乘法是：

$$m_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$m_2 = (A_{21} + A_{22}) \times B_{11}$$
$$m_3 = A_{11} \times (B_{12} - B_{22})$$
$$m_4 = A_{22} \times (B_{21} - B_{11})$$
$$m_5 = (A_{11} + A_{12}) \times B_{22}$$
$$m_6 = (A_{21} - A_{11}) \times (B_{22} + B_{21})$$
$$m_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

# Strassen矩阵乘法

于是可得到:

$$C_{11} = m_1 + m_4 - m_5 + m_7$$
$$C_{12} = m_3 + m_5$$
$$C_{21} = m_2 + m_4$$
$$C_{22} = m_1 + m_3 - m_2 + m_6$$

以上计算的正确性很容易验证。Strassen矩阵乘积分治算法中，用了7次对于$n/2$阶矩阵乘积的递归调用和18次$n/2$阶矩阵的加减运算。由此可知，该算法的所需的计算时间$T(n)$满足如下的递归方程:

# Strassen矩阵乘法

$$\begin{cases} T(n) = 7T(n/2) + n^2 & n >= 2 \\ T(1) = 1 \end{cases}$$

其解为$T(n) \in O(n^{\log_2 7}) \approx O(n^{2.807})$。

由此可见，Strassen矩阵乘法的计算时间复杂性比普通矩阵乘法有所改进。

# 分治法解最近点对问题

问题: 给定平面$S$上$n$个点,找其中的一对点,使得在 $n(n-1)/2$个点对中, 该点对的距离最小。

算法思路:

1) $n$较小时直接求($n=2$).

2) 将$S$上的$n$个点分成大致相等的2个子集$S1$和$S2$

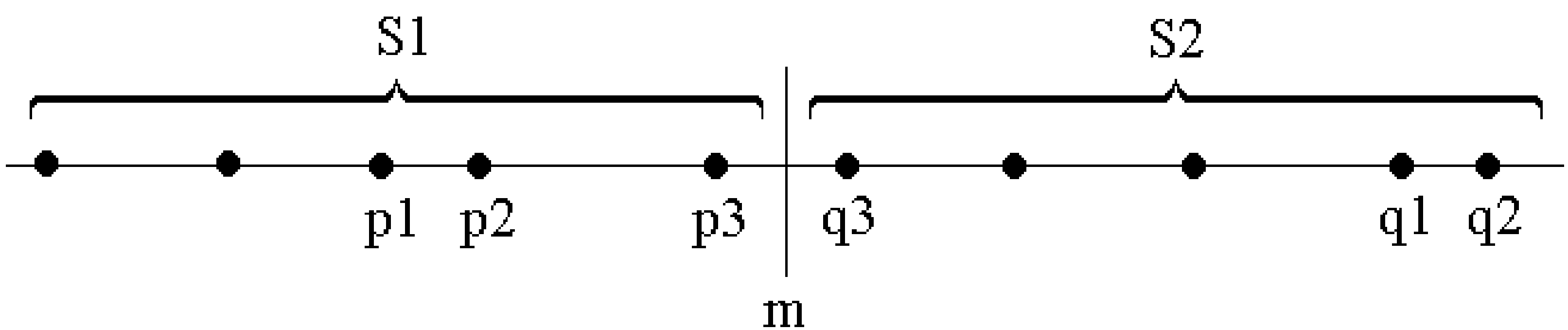
3) 分别求$S1$和$S2$中的最接近点对

4) 求一点在$S1$、另一点在$S2$中的最近点对

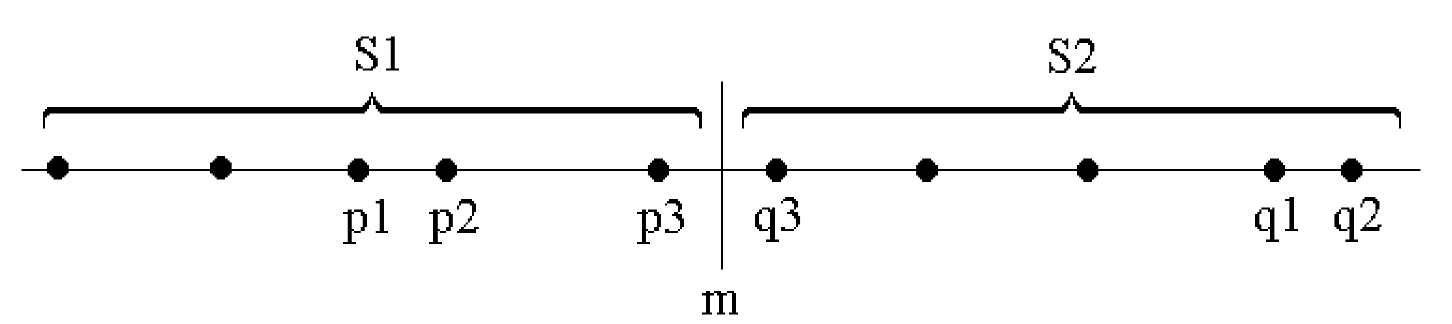
5) 从上述三对点中找距离最近的一对.

# 最近对问题：共线的情况

- 假设我们用x轴上某个点$m$将$S$划分为2个子集$S_1$和$S_2$，基于平衡子问题思想，用$S$中各点坐标的中位数来作分割点。
- 递归地在$S_1$和$S_2$上找出其最接近点对$\{p_1, p_2\}$和$\{q_1, q_2\}$，并设$d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$，$S$中的最接近点对或者是$\{p_1, p_2\}$，或者是$\{q_1, q_2\}$，或者是某个$\{p_3, q_3\}$，其中$p_3 \in S_1$且$q_3 \in S_2$。
- 能否在线性时间内找到$p_3, q_3$?



# 最近对问题：共线的情况

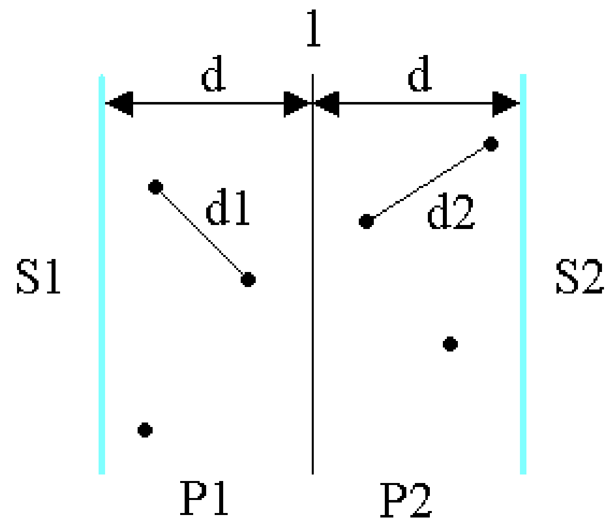



**能否在线性时间内找到$p3, q3$?**

- 如果S的最接近点对是$\{p_3, q_3\}$，即$|p_3 - q_3| < d$，则$p_3$和$q_3$两者与m的距离不超过$d$，即$p_3 \in (m-d, m]$，$q_3 \in (m, m+d]$。
- 由于$S_1$中，每个长度为$d$的半闭区间至多包含一个点（否则必有两点距离小于$d$），并且$m$是$S_1$和$S_2$的分割点，因此$(m-d, m]$中至多包含$S$中的一个点。由图可以看出，如果$(m-d, m]$中有$S$中的点，则此点就是$S_1$中最大点。
- 因此，我们用线性时间就能找到区间$(m-d, m]$和$(m, m+d]$中所有点，即$p_3$和$q_3$。从而我们用线性时间就可以将$S_1$的解和$S_2$的解合并成为$S$的解。

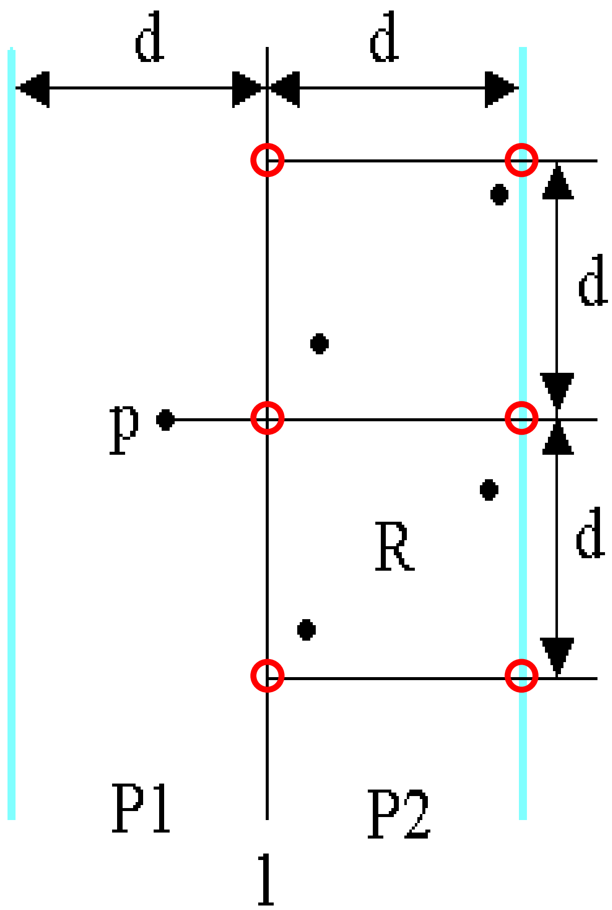
# 最近对问题：二维情形

- 选取一垂直线l:x=m来作为分割直线。其中m为S中各点x坐标的中位数。由此将S分割为$S_1$和$S_2$。
- 递归地在$S_1$和$S_2$上找出其最小距离$d_1$和$d_2$，并设$d=\min\{d_1,d_2\}$，S中的最接近点对或者是d，或者是某个{p,q}，其中$p\in P_1$且$q\in P_2$。
- 能否在线性时间内找到p,q?



# 最近对问题：二维情形

- 考虑$P_1$中任意一点p，它若与$P_2$中的点q构成最接近点对的候选者，则必有distance(p，q) < d。满足这个条件的$P_2$中的点一定落在一个d×2d的矩形R中
- 由d的意义可知，$P_2$中任何2个S中的点的距离都不小于d。由此可以推出矩形R中最多只有6个S中的点。
- 因此，在分治法的合并步骤中最多只需要检查6×n/2=3n个候选者



# 最近对问题

- 为了确切地知道要检查哪6个点，可以将p和$P_2$中所有$S_2$的点投影到垂直线l上。由于能与p点一起构成最接近点对候选者的$S_2$中点一定在矩形R中，所以它们在直线l上的投影点距p在l上投影点的距离小于d。由上面分析可知，这种投影点最多只有6个。
- 因此，若将$P_1$和$P_2$中所有S中点按其y坐标排好序，则对$P_1$中所有点，对排好序的点列作一次扫描，就可以找出所有最接近点对的候选者。对$P_1$中每一点最多只要检查$P_2$中排好序的相继6个点。

# 最近对问题

```
double cpair2(S)
{
    n=|S|;
    if (n < 2) return;
1. m=S中各点x间坐标的中位数;
   //构造S1和S2;
   S1={p∈S|x(p)<=m},
   S2={p∈S|x(p)>m}
2. d1=cpair2(S1);
   d2=cpair2(S2);
3. dm=min(d1,d2);
```

# 最近对问题

4. 设 $P_1$是$S_1$中距垂直分割线l的距离在dm之内的所有点组成的集合；
   $P_2$是$S_2$中距垂直分割线l的距离在dm之内所有点组成的集合；
   将$P_1$和$P_2$中点依其y坐标值排序；
   并设X和Y是相应的已排好序的点列；
5. 扫描X对其每个点检查Y中与其距离dm内所有点(最多6个) 完成合并；
   当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为2dm的区间内移动；
   设dl是按这种扫描方式找到的点对间的最小距离；
6. d=min(dm,dl);
   return d;
}

# 最近对问题

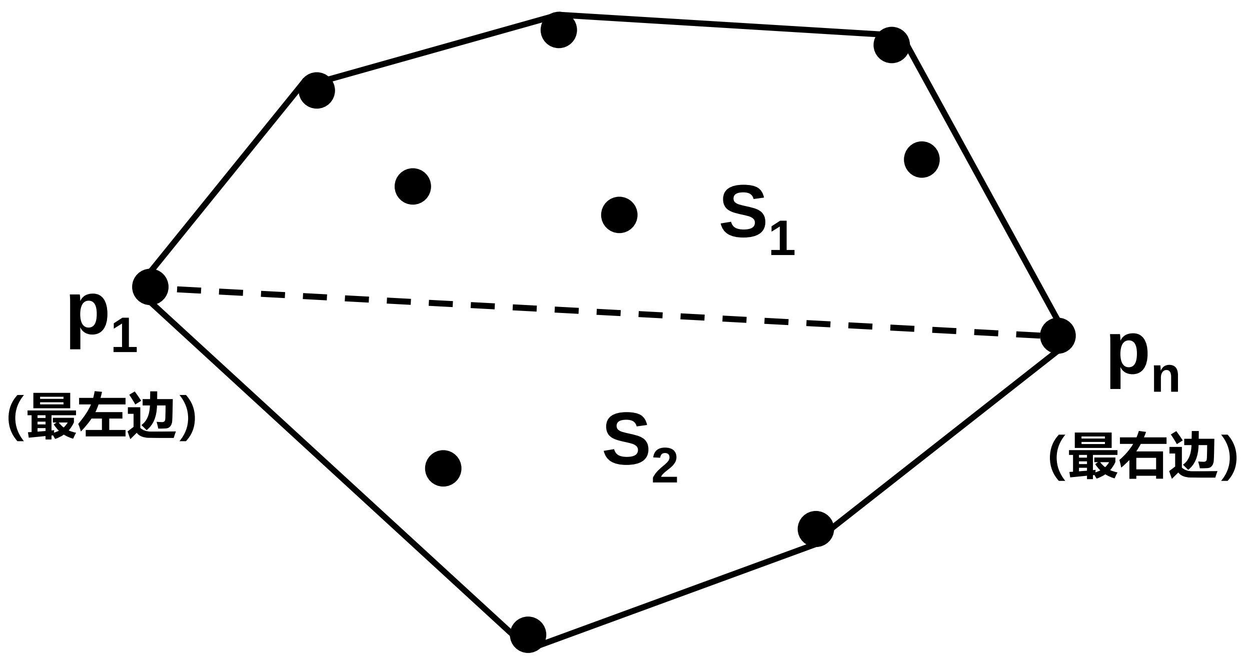
## 复杂度分析

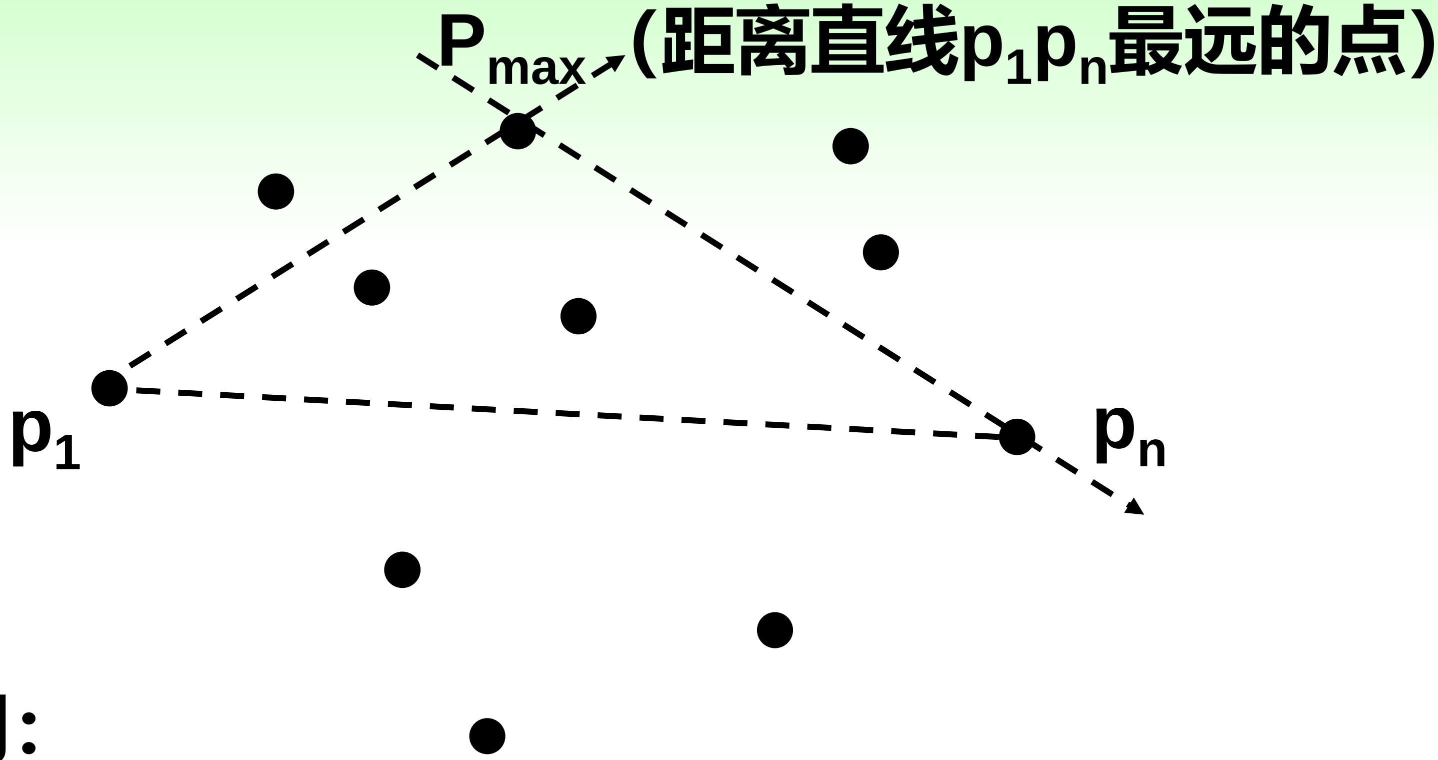
$$T(n)=\begin{cases} O(1) & n<4 \\ 2T(n/2)+O(n) & n\geq 4 \end{cases}$$

$$\boldsymbol{T(n)=O(nlogn)}$$

# 快速凸包算法----分治法求凸包问题

- 快速凸包算法(Quickhull Algorithm)是一个和快速排序(Quicksort Algorithm)相似的分治算法
- 快速凸包算法（**Quickhull Algorithm**）继承了快速排序分治的思想，是一个递归的过程
- **点集合的上包和下包**






- **可证明:**
  - $p_{max}$是上包的顶点
  - 包含在三角形$p_1p_{max}p_n$中的点不可能是上包的顶点
  - 不存在同时位于$p_1p_{max}$和$p_{max}p_n$左边的直线

# 快速凸包算法

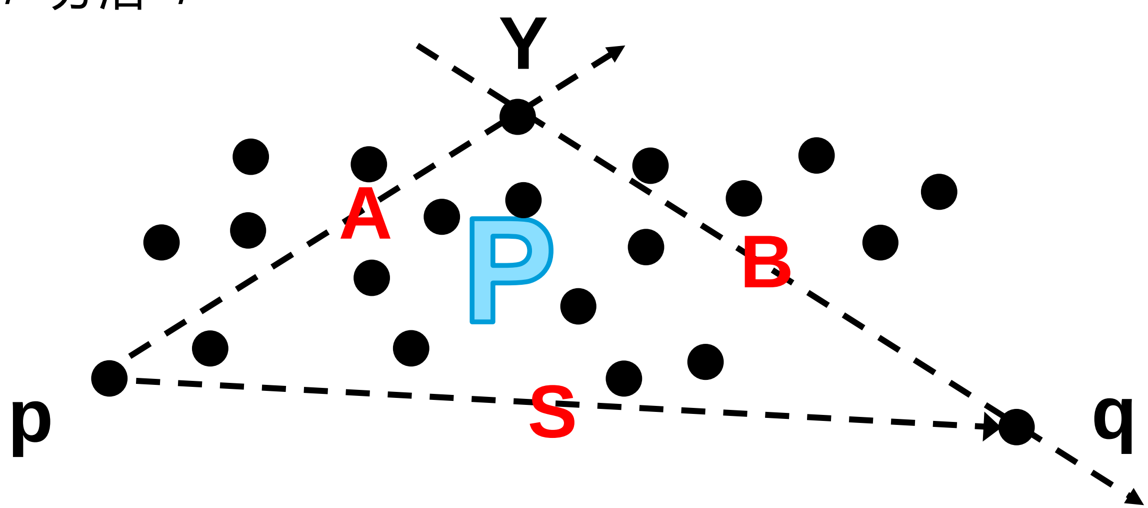
- 对于点$p1 = (x1, y1), p2 = (x2, y2),\ p3 = (x3, y3)$,

  当且仅当下列表达式 为正时，$p3$ 位于直线 $p1p2$ 的左侧。

- $$\begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix} = x1y2 + x3y1 + x2y3 - x3y2 - x2y1 - x1y3$$

# 快速凸包算法

```
void 快速凸包(P:点集 , S:向量 /*S.p,S.q:点*/ ){
        /* P 在 S 左侧，上半个凸包*/
        选取 P 中距离 S 最远的 点 Y ;
        向量 A ← { S.p , Y } ; 向量 B ←{ Y , S.q } ;
        点集 Q ← 在 P 中 且在 A 左侧的点 ;
        点集 R ← 在 P 中 且在 B 左侧的点 ; /* 划分 */
        快速凸包 ( Q , A ) ; /* 分治 */
        输出 (点 Y) ; /* 按中序输出 保证顺序*/
        快速凸包 (R , B ) ; /* 分治 */
}
```



# 快速凸包算法

- 快速凸包算法可达到$O(N^2)$的复杂度，但这需要刻意针对程序经过分析并构造数据能做到，是实际应用中很难碰到的情况
- 在点集均匀分布时快速凸包的复杂度更是达到了$O(N)$ 是其他两种算法难以企及的
- 在绝大多数情况下平均复杂度是$O(N\log_2 N)$也很高效