分治策略
的设计思想

分治策略的基本思想

分治策略（ Divide and Conquer ）

1. 将原始问题划分或者归结为规模较小的子问题
1. 递归或迭代求解每个子问题
2. 将子问题的解综合得到原问题的解

注意:

1. 子问题与原始问题性质完全一样
2. 子问题之间可彼此独立地求解
3. 递归停止时子问题可直接求解

二分检索

算法 Binary Search (*T, l, r, x*)

输入：数组 *T*，下标从 *l* 到 *r*；数 *x*

输出：*j* // 若*x*在*T* 中, *j* 为下标; 否则为 0

```
l←1;  r←n
while l≤r do
    m←⌊(l +r)/2⌋   // m为中间位置
    if T[m]=x then return m // x是中位数
    else if T[m]>x then r←m–1
         else l←m+1
return 0
```

二分检索算法设计思想

- 通过 x 与中位数的比较，将原问题归结为规模减半的子问题，如果 x 小于中位数，则子问题由小于 x 的数构成，否则子问题由大于 x 的数构成.
- 对子问题进行二分检索.
- 当子问题规模为 1 时，直接比较 x与 $T[m]$，若相等则返回 m，否则返回 0.

是否能够递归实现?

二分检索时间复杂度分析

二分检索问题最坏情况下时间复杂度

$$W(n) = W(\lfloor n/2 \rfloor)+1$$

$$W(1) = 1$$

可以解出

$$W(n)=\lfloor \log n \rfloor + 1$$

二分归并排序

算法 **Merge Sort** (A, p, r)

输入：数组 $A[p..r]$

输出：元素按从小到大排序的数组 A

1. **if** $p < r$
2. **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$ 对半划分
3. **Merge Sort** (A, p, q) 子问题1
4. **Merge Sort** ($A, q+1, r$) 子问题 2
5. **Merge** (A, p, q, r) 综合解

二分归并排序设计思想

- 划分将原问题归结为规模为 $n/2$ 的 2 个子问题
- 继续划分，将原问题归结为规模为 $n/4$ 的 4 个子问题. 继续…， 当子问题规模为1 时，划分结束.
- 从规模 1到 $n/2$，陆续归并被排好序的两个子数组. 每归并一次，数组规模扩大一倍，直到原始数组.

二分归并排序时间复杂度分析

假设n为2的幂，二分归并排序最坏情况下时间复杂度

$$W(n) = 2W(n/2) + n-1$$

$$W(1) = 0$$

可以解出

$$W(n) = n\log n - n + 1$$

Hanoi塔的递归算法

算法 **Hanoi** (A, C, n) // n个盘子A到C

1. **if** n=1 **then move** (A, C) //1个盘子A到C
2. **else Hanoi** (A, B, n–1)
3. **move** (A, C)
4. **Hanoi** (B, C, n–1)

设 n个盘子的移动次数为 $T(n)$

$$T(n) = 2\,T(n-1) + 1,$$
$$T(1) = 1,$$
$$T(n)=2^n-1$$

算法设计思想

- 将原问题归结为规模为 n-1 的2个子问题.
- 继续归约，将原问题归结为规模为 n-2 的 4 个子问题. 继续...， 当子问题规模为1 时，归约过程截止.
- 从规模 1到 n-1，陆续组合两个子问题的解. 直到规模为n.

小结

通过几个例子展示分治算法的特点:

- 将原问题归约为规模小的子问题，子问题与原问题具有相同的性质.
- 子问题规模足够小时可直接求解.
- 算法可以递归也可以迭代实现.
- 算法的分析方法：递推方程.

分治算法的一般描述和分析方法

分治算法的一般性描述

分治算法 **Divide-and-Conquer(P)**

1. **if** $|P| \leq c$ **then** $S(P)$
2. **divide** P **into** $P_1, P_2, \ldots, P_k$ 划分
3. **for** $i \leftarrow 1$ **to** k 求解子问题
4. $\quad y_i \leftarrow$ **Divide-and-Conquer(P_i)**
5. **Return Merge** $(y_1, y_2, \ldots, y_k)$ 综合解

设计要点

- 原问题可以划分或者归约为规模较小的子问题
  - 子问题与原问题具有相同的性质
  - 子问题的求解彼此独立
  - 划分时子问题的规模尽可能均衡
- 子问题规模足够小时可直接求解
- 子问题的解综合得到原问题的解
- 算法实现：递归或迭代

分治算法时间分析

时间复杂度函数的递推方程

$$W(n)=W(|P_1|)+W(|P_2|)+\ldots+W(|P_k|)+f(n)$$
$$W(c)=C$$

- $P_1, P_2, \ldots, P_k$ 为 划分后产生的 子问题
- $f(n)$为划分子问题以及将子问题的解综合得到原问题解的总工作量
- 规模为c的最小子问题的工作量为C

两类常见的递推方程

$$f(n)=\sum_{i=1}^{k}a_i f(n-i)+g(n) \qquad (1)$$

$$f(n)=af(\frac{n}{b})+d(n) \qquad (2)$$

例子：

Hanoi塔， $W(n)=2W(n-1)+1$

二分检索，$W(n)=W(n/2)+1$

归并排序，$W(n)=2W(n/2)+n-1$

递推方程的求解

方程1 $$f(n)=\sum_{i=1}^{k}a_i f(n-i)+g(n)$$

方程2 $$f(n)=af(\frac{n}{b})+d(n)$$

求解方法

方程1：迭代法、递归树

方程2：迭代法、换元法、递归树、
主定理

方程2的解

方程 $$T(n)=aT(n/b)+d(n)$$

d(n)为常数

$$T(n)=\begin{cases} O(n^{\log_b a}) & a\neq 1 \\ O(\log n) & a=1 \end{cases}$$

$d(n)=cn$

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

小结

- 分治算法的一般描述

  划分或归约为彼此独立的子问题
  分别求解每个子问题
  给出递归或迭代计算的终止条件
  如何由子问题的解得到原问题解

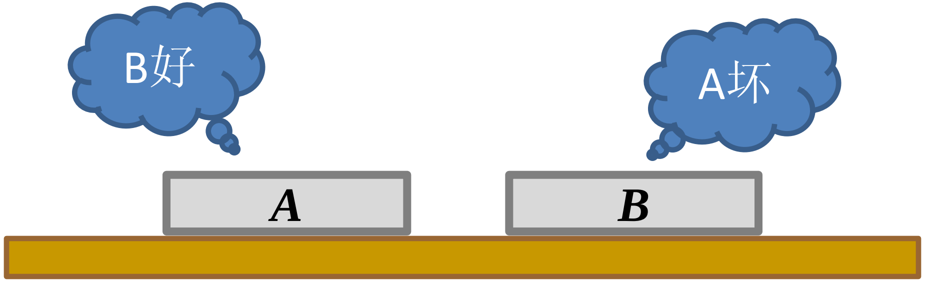
- 分治算法的分析方法

  求解时间复杂度的递推方程
  常用的递推方程的解

芯片测试

一次测试过程

测试方法：将2片芯片（A和B）置于测试台上，互相进行测试，测试报告是“好”或“坏”，只取其一.



假设：好芯片的报告一定是正确的，坏芯片的报告是不确定的（可能会出错）

测试结果分析

A 报告	B 报告	结论
B是好的	A是好的	A, B 都好或 A, B 都坏
B是好的	A是坏的	至少一片是坏的
B是坏的	A是好的	至少一片是坏的
B是坏的	A是坏的	至少一片是坏的

问题

输入：

n 片芯片，其中好芯片至少比坏芯片多 1片.

问题：

设计一种测试方法，通过测试从 *n* 片芯片中挑出 1 片好芯片.

要求：使用最少的测试次数.

判定芯片*A* 的好坏

问题：给定芯片*A*，判定*A*的好坏

方法：用其他 n-1片芯片对 *A* 测试 .

n=7：好芯片数 ≥ 4.

A 好，6个报告中至少 3 个报“好”

A 坏，6个报告中至少 4 个报“坏”

n是奇数：好芯片数 ≥ $(n+1)/2$.

A 好，至少有 $(n-1)/2$个报“好”

A 坏，至少有 $(n+1)/2$个报告“坏”

结论：至少一半报“好”，*A*是好芯片，

超过一半报“坏”，*A*是坏芯片.

判定芯片*A* 的好坏

***n*=8**：好芯片数 ≥ **5.**

_A_ 好，**7**个报告中至少 **4** 个报"好"

_A_ 坏，**7**个报告中至少 **5** 个报"坏"

n是偶数：好芯片数 ≥ ***n*/2+1.**

_A_ 好，至少有 ***n*/2**个报告"好"

_A_ 坏，至少有 ***n*/2+1**个报告"坏"

结论：***n*-1** 份报告中，

至少一半报"好"，则 ***A*** 为好芯片

超过一半报"坏"，则 ***A***为坏芯片

蛮力算法

测试方法：任取 1片测试，如果是好芯片，测试结束；如果是坏芯片，抛弃，再从剩下芯片中任取 1片测试，直到得到 1片好芯片.

时间估计：

第1片坏芯片，最多测试 n-2次，

第2片坏芯片，最多测试 n-3次，

…

总计 $\Theta(n^2)$

分治算法设计思想

假设 n为偶数，将 n片芯片两两一组做测试淘汰，剩下芯片构成子问题，进入下一轮分组淘汰.

淘汰规则：

"好, 好" $\Rightarrow$ 任留 1片，进入下轮
其他情况 $\Rightarrow$ 全部抛弃

递归截止条件：$n \leq 3$

3 片芯片，1 次测试可得到好芯片.

1 或 2 片芯片，不再需要测试.

分治算法的正确性

命题1 当 n 是偶数时，在上述淘汰规则下，经过一轮淘汰，剩下的好芯片比坏芯片至少多1片.

证 设A, B都好的芯片 i 组, A与B一好一坏 j 组，A与B 都坏的 k 组. 淘汰后好芯片至少 i片，坏芯片至多 k片 .

$$2i + 2j + 2k = n$$ 初始芯片总数

$$2i + j > 2k + j$$ 初始 好芯片多于坏芯片

$$\Rightarrow i > k$$

n为奇数时的特殊处理

当 n 是奇数时，可能出问题

输入：好 好 好 好 坏 坏 坏

分组：好 好 | 好 好 | 坏 坏 | 坏

淘汰后：好 好 坏 坏

处理办法：当 n 为奇数时，增加一轮对轮空芯片的单独测试.

如果该芯片为好芯片，算法结束；如果是坏芯片，则淘汰该芯片.

伪码描述

```
算法 Test(n)
1． k ← n
2． while k >3 do
3．     将芯片分成⌊k/2⌋组   // 轮空处理
4．     for i =1 to ⌊k/2⌋ do
5．         if 2片好 then 则任取1片留下
6．         else 2片同时丢掉
7．      k←剩下的芯片数
8． if k =3 then
9．      任取2片芯片测试
10．      if 1好1坏 then 取没测的芯片
11．      else 任取1片被测芯片
12  if k =2 or 1 then 任取1片
```

分组淘汰

递归结束

时间复杂度分析

设输入规模为n

每轮淘汰后，芯片数至少减半

测试次数(含轮空处理)：$O(n)$

时间复杂度：

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

$$W(3) = 1,\ W(2) = W(1) = 0$$

解得 $W(n) = O(n)$

小结

- 芯片测试的分治算法

  如何保证子问题与原问题性质相同：
  增加额外处理

  额外处理的工作量不改变函数的阶

  时间复杂度为$O(n)$

快速排序

基本思想

- 用首元素 x 作划分标准，将输入数组 A 划分成不超过 x 的元素构成的数组 A_L，大于 x 的元素构成的数组 A_R. 其中 A_L, A_R 从左到右存放在数组 A 的位置.
- 递归地对子问题 A_L 和 A_R 进行排序，直到子问题规模为 1 时停止.

伪码

算法 Quicksort (A, p, r)
输入：数组$A[p..r]$
输出：排好序的数组 A

1. if $p < r$
2. then $q \leftarrow$ Partition (A, p, r)
3. $A[p] \leftrightarrow A[q]$
4. Quicksort $(A, p, q\text{-}1)$
5. Quicksort $(A, q+1, r)$

初始置 p=1, r=n，然后调用上述算法

划分过程

```
Partition (A, p, r)
x ← A[p]
i ← p
j ← r + 1
while true do
      repeat  j ← j –1
      until  A[ j ]≤ x  // 不超过首元素的
      repeat  i ← i +1
      until  A[ i ]>x  // 比首元素大的
      if  i < j
      then  A[ i ] ↔ A[ j ]
      else  return j
```

划分实例

27 99 0 8 13 64 86 16 7 10 88 25 90

i *j*

27 25 0 8 13 64 86 16 7 10 88 99 90

i *j*

27 25 0 8 13 10 86 16 7 64 88 99 90

i *j*

27 25 0 8 13 10 7 16 86 64 88 99 90

j *i*

16 25 0 8 13 10 7 27 86 64 88 99 90

时间复杂度

最坏情况: $W(n) = W(n-1)+n-1$

$W(1) = 0$

$W(n) = n(n-1)/2$

最好划分: $T(n) = 2\,T(n/2)+n-1$

$T(1) = 0$

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

均衡划分的时间复杂度

均衡划分：子问题的规模比不变
例如为 1:9

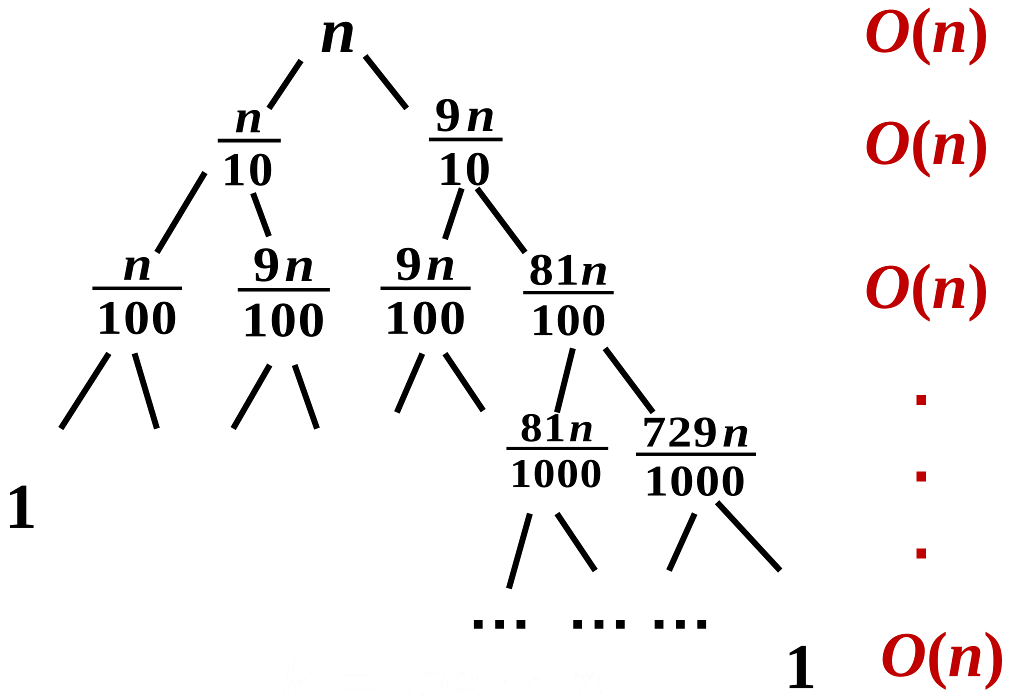
$$T(n) = T(n/10) + T(9n/10) + n$$

$$T(1) = 0$$

根据递归树，时间复杂度

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

递归树



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

平均时间复杂度

首元素排好序后处在 $1, 2, \ldots, n$

各种情况概率都为 $1/n$

首元素在位置 1： $T(0), T(n-1)$

首元素在位置 2： $T(1), T(n-2)$

....

首元素在位置 n-1： $T(n-2), T(1)$

首元素在位置 n： $T(n-1), T(0)$

子问题工作量 $2[T(1)+T(2)+\ldots+T(n-1)]$

划分工作量 $n-1$

平均时间复杂度

$$T(n)=\frac{1}{n}\sum_{k=1}^{n-1}(T(k)+T(n-k))+n-1$$

$$\boxed{\begin{aligned}&T(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+n-1\\&T(1)=0\end{aligned}}$$

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

首元素划分后每个位置概率相等

小结

快速排序算法

- 分治策略
- 子问题划分是由首元素决定
- 最坏情况下时间$O(n^2)$
- 平均情况下时间为$O(n\log n)$

幂乘算法及应用

幂乘问题

输入： a为给定实数，n为自然数

输出： a^n

传统算法： 顺序相乘

$$a^n = (\ldots(((a\ a)a)a)\ldots)a$$

乘法次数： $\Theta(n)$

分治算法——划分

n为偶数 $\underbrace{a\,.\,.\,.\,.\,.\,.\,.\,a}_{n/2\text{个}} \mid \underbrace{a\,.\,.\,.\,.\,.\,.\,.\,a}_{n/2\text{个}}$

n为奇数 $\underbrace{a\,.\,.\,.\,.\,.\,.\,.\,a}_{(n\text{-}1)/2\text{个}} \mid \underbrace{a\,.\,.\,.\,.\,.\,.\,.\,a}_{(n\text{-}1)/2\text{个}} \mid a$

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & n \text{ 为偶数} \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & n \text{ 为奇数} \end{cases}$$

分治算法分析

以乘法作为基本运算

- 子问题规模：不超过$n/2$
- 两个规模近似 $n/2$的子问题完全一样，只要计算1次

$$W(n) = W(n/2) + \Theta(1)$$

$$W(n) = \Theta(\log n)$$

幂乘算法的应用

Fibonacci数列： 1, 1, 2, 3, 5, 8, 13, 21, …

增加 F_0=0，得到数列

0, 1, 1, 2, 3, 5, 8, 13, 21, …

问题: 已知 $F_0=0, F_1=1$, 给定n, 计算 F_n.

通常算法：从 $F_0, F_1, \ldots$ 开始，根据递推公式

$$\boxed{F_n = F_{n-1} + F_{n-2}}$$

陆续相加可得 F_n， 时间复杂度为$\Theta(n)$

Fibonacci数的性质

定理1 设 $\{F_n\}$为 Fibonacci 数构成的数列，那么

$$\boxed{\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n}$$

归纳证明

$n=1$，左边$=\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$=右边

Fibonacci数的性质(续)

假设对任意正整数 n，命题成立，即

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

那么 $$\begin{bmatrix} \boxed{F_{n+2}} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1}$$

归纳假设代入

算法

令矩阵 $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$, 用乘幂算法计算$M^n$

时间复杂度：

- 矩阵乘法次数　$T(n) = \Theta(\log n)$
- 每次矩阵乘法需要做 8 次元素相乘
- 总计元素相乘次数为 $\Theta(\log n)$

小结

- 分治算法的例子——幂乘算法
- 幂乘算法的应用

  计算Fibonacci数

  通常算法$O(n)$，分治算法为$O(\log n)$

改进分治算法的途径1：减少子问题数

减少子问题个数的依据

分治算法的时间复杂度方程

$$W(n) = aW(n/b) + d(n)$$

a：子问题数，n/b：子问题规模，
$d(n)$：划分与综合工作量.

当 a 较大, b较小, $d(n)$不大时,方程的解：

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

减少a是降低函数$W(n)$的阶的途径.

利用子问题的依赖关系，使某些子问题的解通过组合其他子问题的解而得到.

例1：整数位乘问题

输入：X ,Y 是 n 位二进制数， $n = 2^k$

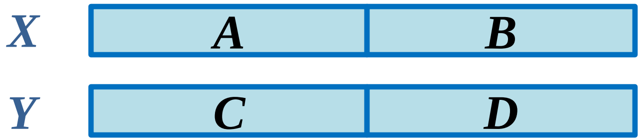
输出：XY

普通乘法：需要$O(n^2)$次位乘运算

简单划分：令

$$X = A2^{n/2} + B, \quad Y = C2^{n/2} + D.$$

$$XY = \underline{AC}\,2^n + (\underline{AD} + \underline{BC})\,2^{n/2} + \underline{BD}$$



$$W(n) = 4W(n/2) + O(n) \Rightarrow W(n) = O(n^2)$$

减少子问题个数

子问题间的依赖关系：代数变换

$$AD+BC = \underline{(A - B)(D - C)} + \underline{AC} + \underline{BD}$$

算法复杂度

$$W(n) = 3\, W(n/2) + cn$$

$$W(1) = 1$$

方程的解

$$W(n) = O(n^{\log 3}) = O(n^{1.59})$$

例2：矩阵相乘问题

输入：A, B 为 n 阶矩阵，$n=2^k$

输出：$C = AB$

通常矩阵乘法：

C 中有 n^2个元素

每个元素需要做 n 次乘法

以元素相乘为基本运算

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

简单分治算法

分治法　将矩阵分块，得

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

其中

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

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

递推方程 $W(n) = 8\, W(n/2) + cn^2$

$W(1) = 1$

解　$W(n) = O(n^3)$.

Strassen 矩阵乘法

变换方法：

设计 $M_1, M_2, \ldots, M_7$，对应7个子问题

$$M_1 = A_{11}(B_{12} - B_{22})$$

$$M_2 = (A_{11} + A_{12})B_{22}$$

$$M_3 = (A_{21} + A_{22})B_{11}$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

Strassen 矩阵乘法(续)

利用中间矩阵，得到结果矩阵

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

时间复杂度函数：

$$\boxed{\begin{array}{l} W(n) = 7\,W(n/2) + 18(n/2)^2 \\ W(1) = 1 \end{array}}$$

解 $W(n) = O(n^{\log 7}) = O(n^{2.8075})$

矩阵乘法的研究及应用

矩阵乘法问题的难度：

- **Coppersmith–Winograd算法：$O(n^{2.376})$** 目前为止最好的上界
- 目前为止最好的下界是： $\Omega(n^2)$

应用：

- 科学计算、图像处理、数据挖掘等
- 回归、聚类、主成分分析、决策树等挖掘算法常涉及大规模矩阵运算

改进途径小结

- 适用于：子问题个数多，划分和综合工作量不太大，时间复杂度函数

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

- 利用子问题依赖关系，用某些子问题解的代数表达式表示另一些子问题的解，减少独立计算子问题个数.

- 综合解的工作量可能会增加，但增加的工作量不影响 $W(n)$的阶.

改进分治算法的途径2：增加预处理

例子：平面点对问题

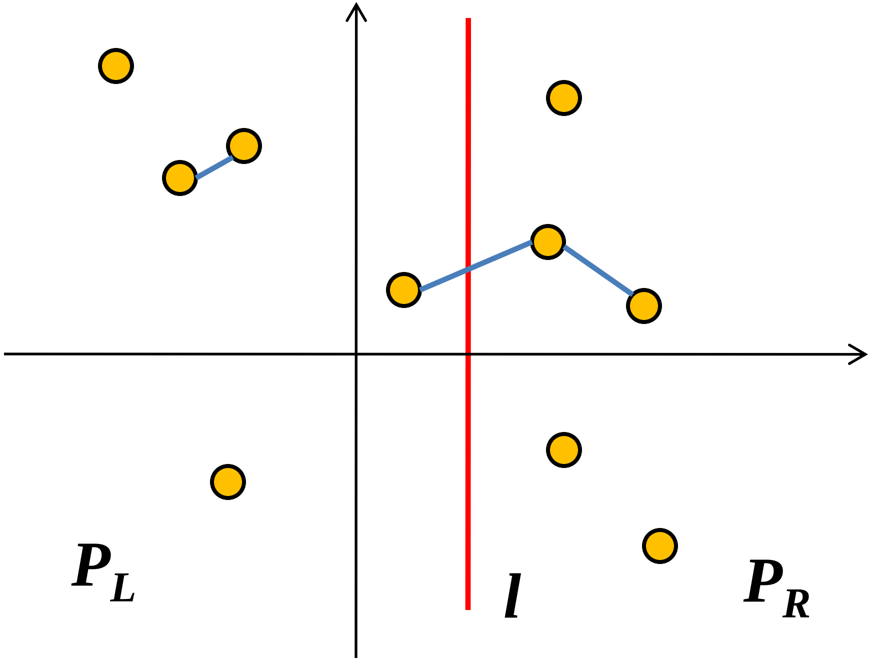
输入: 平面点集 P 中有 n 个点, $n > 1$
输出: P中的两个点，其距离最小

蛮力算法:
$C(n,2)$个点对,计算最小距离，$O(n^2)$

分治策略: P 划为大小相等的P_L和P_R

1. 分别计算 P_L、P_R中最近点对
2. 计算 P_L与P_R中各一个点的最近点对
3. 上述情况下的最近点对是解

划分实例：n=10



算法伪码

MinDistance (P, X, Y)

输入: 点集P, X和Y为横、纵坐标数组

输出: 最近的两个点及距离

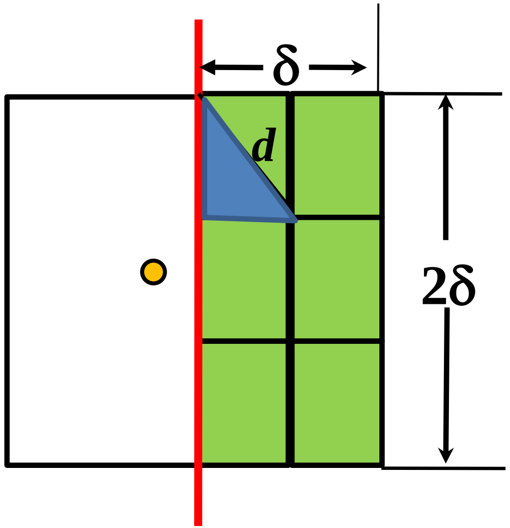
1. 若$|P|\le 3$，直接计算其最小距离
2. 排序X, Y
3. 做中垂线 l 将P划分为P_L和P_R
4. **MinDidtance** (P_L, X_L, Y_L)
5. **MinDistance** (P_R, X_R, Y_R)
6. $\delta=\min(\delta_L,\delta_R)$//$\delta_L,\delta_R$为子问题的距离
7. 检查距 l 不超过δ两侧各1个点的距离. 若小于δ，修改 δ为这个值

跨边界处理

$$d=\sqrt{(\delta/2)^2+(2\delta/3)^2}$$

$$=\sqrt{\delta^2/4+4\delta^2/9}$$

$$=\sqrt{25\delta^2/36}=5\delta/6$$



右边每个小方格至多1个点，每个点至多比较对面的6个点，检查1个点是常数时间，***O*(*n*)** 个点需要***O*(*n*)**时间

算法分析

步1 递归边界处理： $O(1)$

步2 排序： $O(n\log n)$

步3 划分： $O(1)$

步4-5子问题： $2T(n/2)$

步6确定δ: $O(1)$

步7检查跨边界点对： $O(n)$

$$T(n)=2T(n/2)+O(n\log n)$$
$$T(n)=O(1),\ n\le 3$$

递归树求解 $T(n)=O(n\log^2 n)$

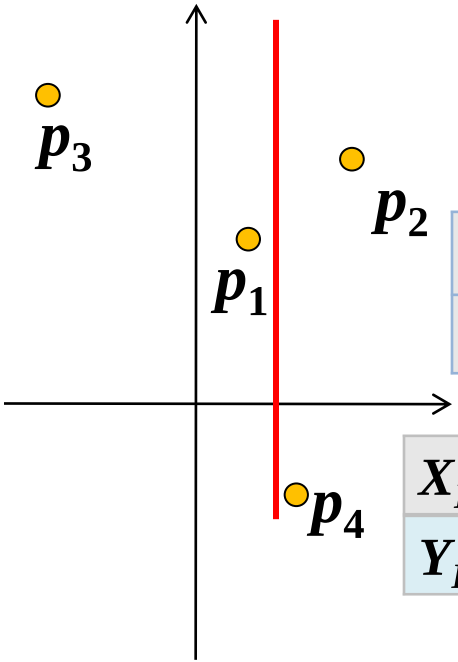
增加预处理

原算法：
在每次划分时对子问题数组重新排序

改进算法：
1. 在递归前对 X,Y 排序，作为预处理
2. 划分时对排序的数组 X,Y 进行拆分，
   得到针对子问题 P_L 的数组 X_L,Y_L 及
   针对子问题 P_R 的数组 X_R,Y_R

原问题规模为 n，拆分的时间为 $O(n)$

实例：递归中的拆分



P	1	2	3	4
X	0.5	2	-2	1
Y	2	3	4	-1

输入

X	-2(3)	0.5(1)	1(4)	2(2)
Y	-1(4)	2(1)	3(2)	4(3)

预处理

X_L	2(3)	0.5(1)
Y_L	2(1)	4(3)

X_R	1(4)	2(2)
Y_R	-1(4)	3(2)

拆分后

改进算法时间复杂度

$W(n)$为算法时间复杂度

递归过程: $T(n)$， 预处理: $O(nlogn)$

$$\boxed{\begin{array}{l} W(n) = T(n) + O(nlogn) \\ T(n) = 2T(n/2) + O(n) \\ T(n) = O(1) \qquad n \le 3 \end{array}}$$

解得 $T(n) = O(nlogn)$

于是 $W(n) = O(nlogn)$

小结

- 依据

$$W(n) = aW(n/b) + f(n)$$

- 提高算法效率的方法：
  - 减少子问题个数a：

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

  - 增加预处理，减少$f(n)$