

Copyright@潘禧辰

contact: panxichen.pxc@gmail.com

全部资料来源任庆生老师ren-qz@cs.sjtu.edu.cn

Ch 01 引论

算法

无论是广义的算法还是狭义的算法，都应满足以下4条性质：

- 有零个或多个输入；（算法可以没有输入）
- 至少有一个输出结果；（算法必须要有输出）
- 构成算法的步骤必须清晰明确；
- 算法能够在有限时间内完成。

算法与编程

- 一台计算机可接受的指令应是定义明确、长度有限的基本操作序列
- 将通常的命令转换为计算机可以理解的指令是一个困难的过程，该过程就是编程。

程序

- 程序是算法用某种程序设计语言的具体体现，可以在计算机上运行，而且还可以不满足算法的第四条限制，即时间有限性。
 - 银行ATM机上运行的程序就是一个在无限循环中执行的程序，不会自动结束。但是其执行的各项任务都是由系统中的一个子程序根据某个特定算法实现的，如查询、取款等，每一个子程序得到结果后便会终止。

Ch 02 数学归纳法

基本原理

- 如果对于带有参数n的命题P，当n=1时P成立；
- 对每一个n，n>1，若n-1时P成立可推出n时P也成立；
- 结论：那么对任意自然数，P都成立。

强数学归纳法

- 如果对于带有参数n的命题P，当n=1时P成立；
- 对每一个n，n>1，若对任意小于n的自然数P成立能推出对n命题P也成立；
- 结论：对任意自然数，P都成立。

变形 (2)

- 如果对于带有参数n的命题P，当n=1和n=2时P都成立；
- 对每一个n，n>2，若n-2时P成立能推出n时P也成立；
- 结论：对任意自然数，P都成立。

变形 (3)

- 如果对于带有参数n的命题P，当n=1时P成立；
- 对每一个n（n大于1且是2的整数幂），若n/2时命题P成立能推出对n命题P也成立；
- 结论：对任意一个2的整数幂的自然数，P都成立。

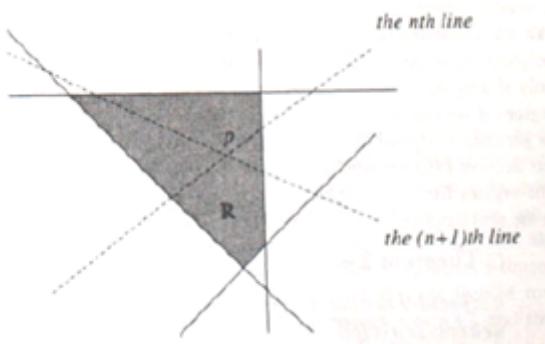
对什么进行归纳

- 在大多数情况下，可对衡量问题规模的n进行归纳
- 在没有现成的可以用来衡量问题规模的参数时，为了使用归纳法，必须构造出一个
- 常用的思路就是考虑如何把命题从小的结构拓展到大的结构。

例子

平面内区域计数

平面上n条居一般位置的直线能把平面分割成多少块区域？



- 分析得到 $D(n) - D(n-1) = n$
-

评述

- 归纳假设处理的是所求函数的增长率，而不是直接对函数本身进行处理；
- 同样的归纳假设用在两个不同的情况下；
- 先猜测再证明

简单的着色问题

证明平面上任意条直线构成的区域可以仅使用两种颜色有效着色，使得相邻的两个区域不同色。

- 数学归纳法第n条直线可以
- 证明反转新加入直线一侧着色即可实现
- 加入 l_{n+1}
 - 对于 l_{n+1} 一侧相邻的区域，原先就颜色不同，反转之后颜色仍然不同
 - 对于 l_{n+1} 两侧相邻的区域，原先颜色相同，现在被 l_{n+1} 分开，一侧反转之后两区域颜色不同。

评述：

- 这个例子告诉我们的基本方法就是如何寻求灵活性，即更大的自由度。解题的思想就是要尽可能地拓展并充分利用归纳假设。对这道题而言，关键的想法就是已知有效着色的前提下，反转所有区域的颜色得到的仍是有效着色，从而解决添加直线后产生新的区域的着色问题的。

复杂加法题

$$1 = 1$$

$$3 + 5 = 8$$

$$7 + 9 + 11 = 27$$

$$13 + 15 + 17 + 19 = 64$$

$$21 + 23 + 25 + 27 + 29 = 125$$

证明：上述三角形中第*i*行的和是 i^3 。

- 转换为增长率 $S(i+1) - S(i) = (i+1)^3 - i^3$
- $2i^2$
 - 第*i+1*行第1个数比第*i*行第1个数大 $2i$
 - 第*i+1*行第2个数比第*i*行第2个数大 $2i$
 - 第*i+1*行第3个数比第*i*行第3个数大 $2i$
 - 第*i+1*行第*i*个数比第*i*行第*i*个数大 $2i$
- 证明第*i+1*行第*i+1*个数大小是 $i^2 + 3i + 1$
- 转化为证明第*i+1*行最后1个数比第*i*行最后1个数大 $2i + 2$
 - 第*i+1*行第*i*个数比第*i*行第*i*个数大 $2i$
 - 第*i+1*行第*i+1*个数比第*i+1*行第*i*个数大2
 - 得证

评述

评述：

- 这段证明告诉我们，整个证明过程不必在一步内完成。只要能取得进展，分步递进就不失为一个好办法。这段证明还体现了“逆向而行”的方法，我们可以从最后的问题开始，通过归约逐步简化问题，而不是从一开始的简单问题开始向最后问题进军。

简单不等式

证明： $\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n} < 1$

- $n = 1, \frac{1}{2} < 1$
- 前*n*项 < 1

$$\begin{aligned} & \underbrace{\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n}}_{= \frac{1}{2} + \frac{1}{2} \left(\frac{1}{2} + \dots + \frac{1}{2^{n-1}} \right)} + \frac{1}{2^{n+1}} \\ & < \frac{1}{2} + \frac{1}{2} * 1 = 1 \end{aligned}$$

评述：

- 在归纳法证明中不必把最后一项看作第*n+1*项。有时候考察第一项来得更容易。有些情况下可以把最后一项认为是满足特殊性质的特殊项。如果遇到这样的问题，要灵活一些，考察尽可能多的方案。

欧拉公式

证明：任意一张连通平面图的节点数(V)、边数(E)和面数(F)的关系可由公式 $V+F=E+2$ 表示。

- 双重归纳法
- 对F进行归纳（双重归纳法第一重）
- F=1
 - 该图为树
 - 图不含回路
 - 且图为连通平面图
 - 证明对于树有 $V=E+1$
 - 对节点数V进行归纳（双重归纳法第二重）
 - $V=1$
 - $V=n$ 时， $E=n-1$ ，证明当 $V=n+1$ 时成立
 - 存在点v，v只与一条边关联，去掉v和相连的边，有 $V=E+1$ ，成立
- 假设 $F=n$ 时命题成立，当 $F=n+1$ 时
 - 存在面f，f同最外边的面相邻
 - f由回路围城
 - 去掉回路上一条边
 - 保持联通性
 - 少了一条边，少了一个面
 - 此时 $F=n$ ，则加回该条边， $F=n+1$ 成立，得证

评述：

- 本定理有三个参数。证明过程中我们对其中一个参数（面数）进行归纳，而归纳基础的成立则需要对另一个参数（结点数）进行归纳。这段证明说明选择归纳顺序的时候要小心谨慎。有时归纳过程从一个参数转到另一个参数，有时归纳与几个参数同时都有关，有时需要同时对两个不同的参数进行归纳。选择不同的顺序进行归纳，证明的难度也大相径庭。

图论问题

令 $G=(V,E)$ 是一个有向图。证明 G 中存在一个独立集(任意两点都不相邻的点构成的集合) $S(G)$ 使得 G 中的每一个结点都可以从 $S(G)$ 的某一个结点通过一条长度不超过2的路可达。

- 对点数进行归纳
- $n \leq 3$ ，成立
- 设 $|V| < n$
 - $|V| = n$ ，对于任意 $v \in V$ 构造相邻点集

构造邻上集

$$N(v) = \underline{\{v\}} \cup \underline{\{w \in V \mid (v, w) \in E\}} \quad |N(v)| \geq 1$$

$$V - N(v) \rightarrow G' = (V', E')$$

$|V'| < n$ 设 $S(G')$ 是 G' 中符合条件的独立集

1). 若 $S(G') \cup \{v\}$ 还是一个独立集

$$S(G) = \underline{S(G')} \cup \{v\}$$

① $\begin{cases} N(v) : \text{从 } v \text{ 出发通过长边的路径到达} \\ V - N(v) \end{cases}$ 由归纳假设

2) $S(G') \cup \{v\}$ 不是独立集

$$\exists w \in S(G') \quad \frac{(w, v) \in E}{(v, w) \text{ 不存在}}$$

考虑 $N(v)$ 中的之 x

$$w \xrightarrow{\text{长度为 } 2} v \xrightarrow{\text{长度为 } 2} x$$

$$S(G) = S(G')$$

评述：

- 证明中“归约”的量并不是固定的。也就是说，我们可以根据实际问题把问题的规模从 n 缩减到比 n 小的数目，而且这个规模小一些的问题并不是任意一个问题，它与原来那个特定的问题关系非常密切。为了让证明容易一些，我们拿掉了足够多的结点。在此，我们取得了一个很好的平衡：既没有拿掉太多的结点使假设太弱，也没有拿掉太少的结点使假设太强。在许多情况下，找到这个平衡点是证明的关键。请注意：这里用了强归纳法，因为我们需要假设对小于某个数的所有数定理都成立。

在图上寻找无重边的路

令 $G = (V, E)$ 是连通的无向图， O 是 V 中度数为奇数的结点的集合，证明：可以把 O 中的结点分成结点对，在每一对中都能找到连接这两个结点的无重边的路。

- $|O|$ 为偶数
- 对边数进行归纳
- $|E| = 1$, 成立

- $|E| < m$, 命题成立
- 当 $|E| = m$ 时, 设 O 为度数为奇数的点集
 - $O = \emptyset$
 - $O \neq \emptyset$, 对于 O 中任意节点 x, y
 - 因为 G 连通, 所以 xy 间一定有路径存在

去掉此路径 \rightarrow 边数 $< m$ 不能用
归纳假设
 \rightarrow 连通性?

- 去掉连通性, G 由一些连通子图构成, 每个连通子图中度数为奇数的点为偶数个, 在某个连通子图中选两个度数为奇数的点, 这两个点之间有路径, 去掉此路径, 边数降低, 应用归纳假设。

评述:

- 我们通过加强归纳假设来证明定理, 这是一个十分有效的方法, 主要的技巧就是要根据需要来改变归纳假设, 尽管需要证明的东西变多了, 然而由于归纳基础变强了, 因而证明的基础比以前更好了, 这样即使定理因此变强了, 证明却有可能变得简单。

数学平均数和几何平均数

证明: 如果 x_1, x_2, \dots, x_n 都是正数, 则 $(x_1 x_2 \cdots x_n)^{1/n} \leq \frac{x_1 + x_2 + \cdots + x_n}{n}$

- 先证明 $n = 2^k$ 时命题成立
 - $k=0, n=1, x_1 \leq x_1$
 - $k=1, n=2, \sqrt{x_1 x_2} \leq \frac{x_1 + x_2}{2}$
 - 设 $n = 2^k$ 时命题成立
 - 考虑 $2n$ 时情况

$$\begin{aligned}
 & (x_1 \cdots x_{2n})^{1/2n} \\
 &= \sqrt{(x_1 \cdots x_n)^{1/n} (x_{n+1} \cdots x_{2n})^{1/n}} \\
 &\leq \frac{(x_1 \cdots x_n)^{1/n} + (x_{n+1} \cdots x_{2n})^{1/n}}{2} \\
 &\leq \frac{\underbrace{x_1 + \cdots + x_n}_n + \underbrace{\frac{x_{n+1} \cdots x_{2n}}{n}}_2}{2}
 \end{aligned}$$

= 右侧

- 逆向归纳 $n \rightarrow n-1$

$$\bar{x} = \frac{x_1 + \dots + x_{n-1}}{n-1}$$

$x_1, x_2, \dots, x_{n-1}, \bar{x}$ } n个数

$$(x_1, \dots, x_{n-1}, \bar{x})^{\frac{1}{n-1}} \leq \frac{(x_1 + \dots + x_{n-1}) + \bar{x}}{n}$$

$$= \frac{(n-1)\bar{x} + \bar{x}}{n} = \bar{x}$$

$$x_1, \dots, x_{n-1} \leq \bar{x}^{n-1}$$

$$(x_1, \dots, x_{n-1})^{\frac{1}{n-1}} \leq \bar{x} = \frac{x_1 + \dots + x_{n-1}}{n-1}$$

逆向归纳法原理：

- 如果命题P对某个自然数的无穷子集成立，且P对n成立能推出其对n-1成立，那么P对任意自然数都成立。

循环不变量

用以证明主体结构为循环的算法的正确性，与数学归纳法证明命题正确性相似

要点：

- 首先定义算法的循环体所操作的数据或数据结构上的一个关键性质（称为循环不变量）
- 证明该性质在循环体开始之前是成立的
- 证明循环体每次执行之后该性质仍然成立
- 最后证明算法结束时该关键性质保证了算法的正确性

例子

将十进制数转换为二进制数

证明：当算法Convert_to_Binary终止时，n的二进制表示存放在数组b中。

```

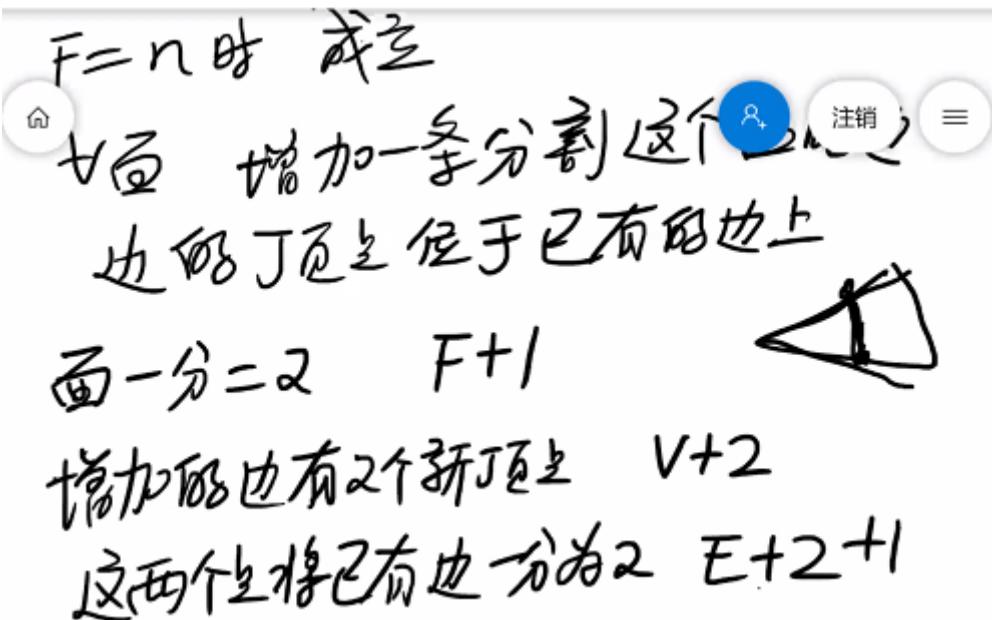
Algorithm Convert_to_Binary(n)
Input: n (a positive integer)
Output: b (an array of bits corresponding to the binary representation of n)
Begin
    t:=n; {use a new variable t to preserve n}
    k:=0;
    while t>0 do
        k:=k+1;
        b[k]:=t mod 2;
        t:=t div 2;
    end

```

- m是由二进制数组b[1.....k] ($b_k b_{k-1} \dots b_1$)表示的二进制整数
- 定义循环不变量: $n = t2^k + m$
- 证明该性质在循环体开始之前是成立的: k=0, m=0, n=t
- 证明循环体每次执行之后该性质仍然成立
 - 第k次循环开始时, $n = t2^k + m$
 - t是偶数
 - $t \bmod 2 = 0$, 对数组无影响
 - t是奇数
 - $b[k+1]=1$, 相当于m增加了 2^k , t变为 $\frac{t-1}{2}$, k变为k+1
 - $\frac{t-1}{2}2^{k+1} + m + 2^k = t2^k + m = n$
- 证明算法结束时该关键性质保证了算法的正确性
 - $t=0$ 循环结束
 - $n = 02^k + m = m$

常见错误

- 如果你对一个定理深信不疑, 往往就会把一些看上去可以从该定理推出的“事实”用在证明中。
- 数学归纳法证明中的主要的一步就是要证明定理对n成立能推得定理对n+1也成立。可以先给出n+1时的情况, 然后证明其可以从n时的情况得到; 也可以先给出n时的情况, 然后证明其能推出n+1时的情况。这两种方法都可以, 然而, n+1时的情况必须是任意的!
 - 例如欧拉公式例题 $V+F=E+2$



- 只证明了特殊情况下的n+1, 并非任意的
- 定理中有特殊情况存在

小结

- 数学归纳法的第一步就是定义归纳假设, 要决定对哪个参数进行归纳。
- 每次用归纳法证明都分成两步: 归纳基础和归约。归纳基础一般都比较容易, 核心是归约, 归约的要点在于要完整地保留原命题, 不能在归约后的命题里加入任何多余的假设, 除非这些假设是特别包含在归纳假设中的。归约也可以被看作是扩展。对某个特定的参数, 把命题从较小的值扩展到较大的值, 必须保证这样的扩展覆盖了参数中所有可能的值, 且扩展后的命题是定理的一般形式, 没有其它多余的假设或限制。

Ch 03 算法分析

算法分析的目的是要预测算法的行为，包括时间复杂性和空间复杂性，从而判断算法的优劣。

时间复杂性

决定程序运行时间的因素：

- 问题规模（输入规模）
- 计算机系统性能（影响较弱）
- 算法的时间复杂性

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
1024	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
2048	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
4096	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
8192	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
16384	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
32768	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
65536	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
131072	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
262144	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
524288	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent
1048576	0.02 μ					

N代表相同时间内求解问题的规模

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

渐进时间复杂性

- 评估算法性能并不需要对其执行时间作出准确统计，关心相对性能
- 在问题规模较小的情况下，不同算法求解同一问题的时间虽然会有差异，但可以忽略不计
- 我们最关心的是大规模问题的性能
- 我们更为关心的是在大规模的输入下时间复杂性增长趋势的差异，因此在比较两个算法时间复杂性时，通常是在输入规模 n 趋向无穷大的情形下进行讨论的，即进行渐近分析。

O

如果存在常数 c 和 N ，使得对于所有的 $n^3 \geq N$ ，有 $g(n) \leq cf(n)$ ，则称函数 $g(n)$ 相对于另一函数 $f(n)$ 是 $O(f(n))$ 的（读作“O”或者“大O”）。对于足够大的 n ，函数 $g(n)$ 不超过函数 $f(n)$ 的一个固定倍数。函数 $g(n)$ 可能比 $cf(n)$ 小，甚至是常数；符号 O 仅仅是从上限来约束它。

$$\begin{aligned} T(n) &= (n+1)^2 \\ &= n^2 + 2n + 1 \\ &\leq n^2 + 2n^2 + n^2 \quad \underline{\underline{n \geq 1}} \\ &= 4n^2 \\ C = 4 \quad N = 1 \quad n \geq N. \\ T(n) &\leq 4 \cdot n^2 \therefore T(n) = O(n^2) \end{aligned}$$

- 无系数
- 无低阶项

$$\begin{aligned} g(n) &= O(\cancel{T(n)}) \quad \cancel{O(3n^2)} \\ g(n) &= O(\cancel{n^2 + n}) = O(n^2) \\ &\quad n^2 \gg n \\ &\quad \text{低阶项} \end{aligned}$$

定理3.1：对于所有的常数 $c>0$ 和 $a>1$ ，以及所有的单调递增函数 $f(n)$ ，有 $(f(n))^c=O(a^{f(n)})$ 。换句话说，一个指数函数要比一个多项式函数增长得快。

- 引理3.2：1) 如果 $f(n)=O(s(n))$ 并且 $g(n)=O(r(n))$ ，则 $f(n)+g(n)=O(s(n)+r(n))$ 。2) 如果 $f(n)=O(s(n))$ 并且 $g(n)=O(r(n))$ ，则 $f(n)g(n)=O(s(n)g(n))$ 。

由于符号 O 对应于关系“ \leq ”，不能对其定义减法和除法运算。

也就是说，通常 $f(n)=O(s(n))$ 和 $g(n)=O(r(n))$ 并不意味 $f(n)-g(n)=O(s(n)-r(n))$ ，或者 $f(n)/g(n)=O(s(n)/r(n))$

Ω

如果存在常数 c 和 N ，使得对于所有的 $n \geq N$ ，有 $g(n) \geq cf(n)$ ，则称函数 $g(n)$ 相对于另一函数 $f(n)$ 是 $\Omega(f(n))$ 的。

$f(n)=\Omega(g(n))$ 当且仅当 $g(n)=O(f(n))$

Θ

如果存在常数 c_1, c_2 和 N ，使得对于所有的 $n \geq N$ ，有 $c_1f(n) \leq g(n) \leq c_2f(n)$ ，则称函数 $g(n)$ 相对于另一函数 $f(n)$ 是 $\Theta(f(n))$ 的。

$f(n)=\Theta(g(n))$ 当且仅当 $f(n)=\Omega(g(n))$, $f(n)=O(g(n))$

←
🔍
注销
≡

$$f(n) = c \quad \text{常数}$$

$$\underline{O(1) \quad \mathcal{R}(1) \quad \mathcal{H}(1)} \quad \checkmark$$

$$f(n) = f(n+1)$$

$$f(n) = 2^n \quad \underline{2^n = \mathcal{H}(2^{n+1})}$$

$$f(n) = n! \quad n! = O((n+1)!) \\ * \underline{\mathcal{H}}$$

$$n! = \underline{\mathcal{R}}(n+1)! \quad \times$$

o

如果对于任意的 $c > 0$, 可以找到 $N > 0$, 使得对于所有的 $n \geq N$, 有 $g(n) < cf(n)$, 则称函数 $g(n)$ 相对于另一函数 $f(n)$ 是 $o(f(n))$

如果 $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$, 则 $f(n) = o(g(n))$

$f(n) = o(g(n))$ 当且仅当 $f(n) = O(g(n))$ 但 $g(n) \neq O(f(n))$

定理3.3: 对于所有的常数 $c > 0$ 和 $a > 1$, 以及所有的单调递增函数 $f(n)$, 有 $(f(n))^c = o(a^{f(n)})$ 。换句话说, 一个指数式函数要比一个多项式函数增长得快。

$$1 \prec \log \log n \prec \log n \prec \sqrt{n} \prec n^{3/4} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$$

- c 为任意

$$f(n) = \frac{1}{2}n^2 \quad O(n^2)$$

$$\Omega(n^2)$$

$$\frac{\frac{1}{2}n^2 < n^2}{\checkmark}$$

$$\forall c \quad \frac{1}{2}n^2 < c \cdot n^2 \quad X$$

$$f(n) = O(n^2)$$

$$\neq \Omega(n^2)$$

$$f(n) = \Omega(n^3)$$

$$\frac{1}{2}n^2 < c \cdot n^3$$

空间复杂度

- 空间复杂度表明的是运行一个算法所需要的临时存储空间。（工作空间而非输入输出空间）
- 不把输入和输出需要的空间作为空间复杂度的一部分，也不对程序自身占用的空间计数
- $S(n)=O(T(n))$ 时间复杂性是空间复杂性的上界
- 时间与空间的平衡

最优算法

通常如果我们能够证明某一问题的时间复杂性为 $\Omega(f(n))$ ，那么任一能够在 $O(f(n))$ 时间内求解该问题的算法即为最优算法

$$P: \Omega(n^2)$$

$$A_1 \quad T_1(n) = 1000n^2 + 2000n + 100$$

$$A_2 \quad T_2(n) = 0.5n^2$$

$$A_3 \quad T_3(n) = n + 1$$

$$A_4 \quad T_4(n) = n^4 + 1$$

最优算法

- A1 ✓
- A2 ✓
- A3错误，不可能被设计出来

如何估计算法运行时间

- 计算迭代次
- 计算基本运算的频度
- 使用递推关系

基本运算

基本运算

所谓基本运算是指具有最高频度的运算，
其它运算的频度最多为基本运算频度的常数倍

Algorithm 1 INSERTIONSORT

Input: An array A[1..n] of n elements

Output: A[1..n] sorted in nondecreasing order

1. for $i \leftarrow 2$ to n
2. $x \leftarrow A[i]$
3. $j \leftarrow i - 1$
4. while ($j > 0$) and ($A[j] > x$)
5. $A[j+1] \leftarrow A[j]$
6. $j \leftarrow j - 1$
7. end while
8. $A[j+1] \leftarrow x$
9. endfor

Algorithm 2 MODINSERTIONSORT

Input: An array A[1..n] of n elements

Output: A[1..n] sorted in nondecreasing order

1. for $i \leftarrow 2$ to n
2. $x \leftarrow A[i]$
3. $k \leftarrow \text{MODBINARYSEARCH}(A[1..i-1], x)$
4. for $j \leftarrow i - 1$ downto k
5. $A[j+1] \leftarrow A[j]$
6. endfor
7. $A[k] \leftarrow x$
8. endfor

15

最坏情况和平均情况分析

- 对很多问题，算法性能不仅取决于输入的规模，也取决于输入的形式
- 最坏情况，平均情况，最优情况

输入规模

•

$\text{for } i=1 \text{ to } n$
 $\quad \text{for } j=1 \text{ to } n$
 $\quad \quad c[i,j] = \underline{a[i,j] + b[i,j]}$
 $\quad \quad \quad H(n^2) = \underline{n^2}$

- 1 线性时间算法 ✓
- 2 平方

- 根据输入规模判断算法复杂度
- 该例中输入为 n^2

$\text{for } i=2 \text{ to } \sqrt{n}$
 if $n \% i = 0$
 输出 n 为合数
 并结束
 end if
 end for
 输出 n 为素数

- 1) 比线性时间还要快
- 2) 线性时间算法
- 3) 多项式时间 --- (大于 $\frac{1}{n^2}$)
- 4) 不是多项式 ---

- 答案为4
- 输入规模为 n 值大小，用比特数衡量
 - 比特数 $\text{bit} = \log n$

- $\sqrt{n} = 2^{\log n/2}$ 对比特数是指数复杂度

对数

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b(c^y) = y \log_b c$$

$$\log_a b = \frac{1}{\log_b a} \quad \log_b x = \frac{\log_a x}{\log_a b}$$

$$a^{\log_a x} = x \quad x^{\log_b y} = y^{\log_b x}$$

$$\sum_{i=1}^n \lfloor \log i \rfloor = (n+1)\lfloor \log n \rfloor - 2^{\lfloor \log n \rfloor + 1} + 2 = \Theta(n \log n)$$

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

顶函数和底函数

若x为实数，则x的底函数 $\lfloor x \rfloor$ 为不大于x的最大整数，x的顶函数 $\lceil x \rceil$ 为不小于x的最小整数

$$\lceil x/2 \rceil + \lfloor x/2 \rfloor = x \quad x \text{ 为整数}$$

$$\lfloor -x \rfloor = -\lceil x \rceil, \lceil -x \rceil = -\lfloor x \rfloor$$

定理：f(x)为单调递增函数，若f(x)是整数时x必为整数，则
 $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$ and $\lceil f(\lceil x \rceil) \rceil = \lceil f(x) \rceil$

定理应用

$$f(x) = x/n \quad n \in N$$

$$\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$$

$$\lfloor \lfloor x \rfloor / n \rfloor = \lfloor x/n \rfloor$$

$$\lfloor \lfloor n/2 \rfloor / 2 \rfloor = \lfloor n/4 \rfloor \quad *$$

$$\underline{n=2^k} \quad \underline{\lfloor n/2 \rfloor} \quad \underline{n/2}$$

阶乘

$$0! = 1$$

$$n! = n(n-1)! \quad n \geq 1$$

Stirling 公式

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n (1 + O(1/n)) \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n$$

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

二项式系数

$$C_n^k = \frac{n!}{k!(n-k)!}$$

$$\binom{n}{k}$$

$$\binom{n}{k} = \binom{n}{n-k} \quad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$(1+x)^n = \sum_{j=0}^n \binom{n}{j} x^j$$

Pascal 三角形

倒数第二个公式

$$\begin{aligned} x=1 & \quad 2^n = \left(\sum_{j=0}^n C_n^j \right) \\ x=-1 & \quad x^j = (-1)^j \quad \sum_{\text{奇}} C_n^j = \sum_{\text{偶}} (-1)^j \end{aligned}$$

鸽巢原理

定理：如果要把n个球放入m个盒子中，则

- 1) 存在一个盒子，至少装了 $\lceil n/m \rceil$ 个球
- 2) 存在一个盒子，至多装了 $\lfloor n/m \rfloor$ 个球

例子

•

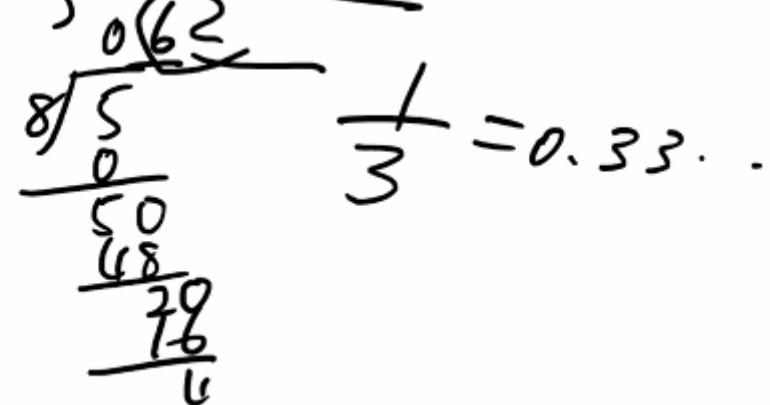
连通无向图 $G = (V, E)$ ↑奇
 $|V| = n$, $|E|: m > n$
 回路

◦ m : 实际走的路数

2) 正整数 $a < b$ 

while ($a > 0$)

{
 输出 a/b (整除)
 $a = (a \% b) * 10$


 $\frac{5}{8} \quad \frac{1}{3} = 0.33\ldots$

$a \% b \in [0, b-1]$

$b+1$ 次后必然出现相同值，有循环小数，可以停掉

和式

$$\begin{aligned}
& \sum_{j=1}^n a_{f(j)} \text{ or } \sum_{1 \leq j \leq n} a_{f(j)} \\
& \sum_{j=1}^n j = \frac{n(n+1)}{2} = \Theta(n^2) \quad \sum_{j=1}^n j^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3) \\
& \sum_{j=0}^n c^j = \frac{c^{n+1}-1}{c-1} = \Theta(c^n) \quad c \neq 1 \\
& \sum_{j=1}^n \frac{1}{j} = \ln n + \gamma + O(1/n) \quad \text{Euler常数 } \gamma = 0.577... \\
& \sum_{j=0}^n 2^j = 2^{n+1} - 1 = \Theta(2^n) \quad \sum_{j=0}^n \frac{1}{2^j} = 2 - \frac{1}{2^n} < 2 = \Theta(1) \\
& \sum_{j=0}^{\infty} c^j = \frac{1}{1-c} = \Theta(1) \quad |c| < 1 \\
& \sum_{j=1}^n jc^j = \frac{nc^{n+2} - nc^{n+1} - c^{n+1} + c}{(c-1)^2} = \Theta(nc^n) \quad c \neq 1 \\
& \sum_{j=1}^n \frac{j}{2^j} = 2 - \frac{n+2}{2^n} = \Theta(1) \quad \sum_{j=0}^{\infty} jc^j = \frac{c}{(1-c)^2} = \Theta(1) \quad |c| < 1
\end{aligned}$$

- 倒数第三个推导-对倒数第四个求导×c

求和的积分近似

$f(x)$ 是单调递减的连续函数，则

$$\int_m^{m+1} f(x) dx \leq \sum_{j=m}^n f(j) \leq \int_{m-1}^m f(x) dx$$

$f(x)$ 是单调递增的连续函数，则

$$\int_{m-1}^m f(x) dx \leq \sum_{j=m}^n f(j) \leq \int_m^{m+1} f(x) dx$$

- 分别取小矩形左边和右边，获得不同上下界

积分近似应用

$$\sum_1^n \log j = \underline{\log n} + \sum_1^{n-1} \log j'$$

$$= \log n + \int_1^n \log x dx$$

$$= \log n + \underline{n \log 1} - n \log e + \log e$$

$$\sum_1^n \log j = \sum_2^n \geq \int_1^n \log x dx$$

$$= \underline{n \log n} - n \log e + \log e$$

递推关系

递推关系 (Recurrence relation, 也称递归关系) 是一种函数定义方式，其在函数的定义体中包含了函数本身。最著名的递推关系可能是Fibonacci数的定义

$$F(n) = F(n-1) + F(n-2), \quad F(1) = 1, \quad F(2) = 1$$

如果有一个 $F(n)$ 的显式(或称闭形式的)表达式，就方便得多了。我们可以快速计算出 $F(n)$ 并与其他函数相比较。这个过程称之为求解递推关系。

- 猜测证明

$$n = 2^k$$

$$T(n) = O(\underline{f(n)})$$

$$\begin{aligned} f(n) &= n^2 \\ \circ 1) \quad T(2) &= 1 \leq f(2) = 4 \\ \circ 2) \quad \text{若 } T(n) \leq n^2 \end{aligned}$$

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2n^2 + \underline{2n - 1} \xrightarrow{\text{由 } 2) \text{ 得}} 2n^2 \\ &\leq (2n)^2 \end{aligned}$$

$$T(n) = \tilde{O}(n^2)$$

$$\circ \quad T(n) = O(n), \quad T(n) \leq Cn$$

若 $f(n) = cn$

$$T(2n) \leq 2T(n) + 2n - 1$$
$$\leq 2cn + \underline{2n-1}$$

$$\underline{T(2n)} \leq c \cdot 2n$$

◦

猜 $T(n) = O(n \log n)$

若 $\underline{T(n) = n \log n}$

$$T(2n) \leq 2T(n) + 2n - 1$$
$$\leq 2n \log n + 2n - 1$$
$$= 2n \log(2n)$$

◦ 常见错误，猜测 $T(n) = O(n)$ ，应该假设 $T(n) \leq Cn$ ，而非 $T(n) = O(n)$

$$T(2n) \leq 2T(n) + 2n - 1$$

$$= O(n) + \underline{2n-1}$$

$$= O(n)$$

猜想 $O(f(n))$ 假设 $T(n) \leq c \cdot f(n)$

• 线性齐次递推式

◦ 线性齐次递推式： $f(n) = a_1f(n-1) + a_2f(n-2) + \dots + a_kf(n-k)$

此时， $f(n)$ 被称为是 k 阶的

求解方法：特征方程

◦

$$1 \text{ 归纳} \quad f(n) = af(n-1) \\ = a^2 f(n-2) \cdots = a^n f(0)$$

$$2 \text{ 递归} \quad f(n) = a_1 f(n-1) + a_2 f(n-2) \\ x^2 = a_1 x + a_2 \\ \text{根为 } r_1, r_2 \text{ (包括虚根)}$$

$$\begin{array}{ll} r_1 \neq r_2 & f(n) = c_1 r_1^n + c_2 r_2^n \\ r_1 = r_2 & f(n) = c_1 r^n + c_2 n \cdot r^n \\ & \overline{f(0)} \quad \overline{f'(1)} \end{array}$$

- 分治关系

在分治法中，问题通常被分割成几个子问题，其中每一个子问题也将被递归求解，得到结果后，再使用一个操作来组合子问题的解以得到原始问题的解。假设有a个子问题，每个问题的规模是原始问题的 $1/b$ ，并且用于组合各子问题解的算法运行时间是 cn^k ，其中a, b, c和k是常数，即

$$T(n) = aT(n/b) + cn^k$$

定理3.4：常数a和b为整数， $a \geq 1$, $b \geq 2$, 且c和k是正常数，则递推关系 $T(n) = aT(n/b) + cn^k$ 的解是

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

$$\begin{aligned}
 T(n) &= a \underline{T(n/b)} + C \cdot n^k \\
 &= a \left(a \underline{T(n/b^2)} + C \cdot (n/b)^k \right) + \dots \\
 &= a \left(a \left(a \underline{T(n/b^3)} + C \cdot (n/b^2)^k \right) + C \cdot (n/b)^k \right) \\
 &\quad + C \cdot n^k \\
 &= a \left(a \left(\dots T(n/b^m) + C \cdot (n/b^{m-1})^k \right) + \dots \right) + C \cdot n^k
 \end{aligned}$$

若 $T(1) = C$

$$\begin{aligned}
 T(n) &= \underline{C \cdot a^m} + C \cdot a^{m-1} \cdot b^k + C \cdot a^{m-2} \cdot b^{2k} \\
 &\quad + \dots + C \cdot b^{mk} \\
 &= C \cdot \sum_{i=0}^m a^{m-i} b^{ik} = C \cdot a^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i
 \end{aligned}$$

1). $a > b^k$ \sum 不超过常数

$$\begin{aligned}
 \therefore T(n) &= O(a^m) \quad m = \log_b n \\
 &= O\left(n^{\log_b a}\right) = O(n^{\log_b a})
 \end{aligned}$$

2) $a = b^k$ $k = \log_b a$ $m = O(\log n)$

$$T(n) = O(m \cdot a^m) = O(n^k \cdot \log n)$$

$$3) a < b^k \quad F = b^k/a$$

$$T(n) = a^m \cdot \frac{F^{m+1} - 1}{F - 1} = O(a^m \cdot F^m)$$

$$= O((b^k)^m) = O((b^m)^k)$$

$$= O(n^k)$$

涉及全部历史的递推关系

- 一个涉及全部历史的递推关系，是依赖于先前所有函数值的函数，而并非仅与一个或者几个有关。
- 历史消除法

eg1

初始条件 $n \geq 2$

$$T(n) = C + \sum_{i=1}^{n-1} T(i)$$

$$T(n+1) = C + \sum_{i=1}^n T(i)$$

$$\underline{T(n+1) - T(n) = T(n)}$$

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

$$= T(1) 2^n$$

注意有效范围

$$\begin{aligned} T(n+1) &= T(2) \cdot 2^{n-1} \\ &= (T(1) + C) \cdot 2^{n-1} \end{aligned}$$

eg2

$$T(n) = (n-1) + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \quad n \geq 2$$

$$T(1) = 0$$

$$T(n+1) = \underbrace{\frac{2}{n+1} \sum_{i=1}^{n-1} T(i)}_{nT(n)} + (n+1) \cdot T(n)$$

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

$$T(n+1) = \frac{n+2}{n+1} T(n) + \frac{2n}{n+1}$$

$$\leq \frac{n+2}{n+1} T(n) + 2$$

$$T(n) \leq \frac{n+1}{n} T(n-1) + 2 \\ \leq 2 + \frac{n+1}{n} \left(2 + \frac{n}{n-1} T(n-2) \right)$$

$$\leq 2 + \frac{n+1}{n} \left(2 + \frac{n}{n-1} \left(2 + \frac{n-1}{n-2} \left(\dots \frac{4}{3} \dots \right) \right) \right)$$

$$= 2 \left(1 + \frac{n+1}{n} + \frac{n+1}{n} \cdot \frac{n}{n-1} + \frac{n+1}{n} \cdot \frac{n}{n-1} \cdot \frac{n-1}{n-2} \right)$$

$$+ \dots + \frac{n+1}{n} \cdot \frac{n}{n-1} \cdots \frac{4}{3} \right)$$

$$= 2 \left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \dots + \frac{n+1}{3} \right)$$

$$\begin{aligned}
 &= 2(n+1) \left(\underbrace{\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}}_{\approx \ln n + \gamma - 1.5} \right) \\
 \therefore T(n) &\leq 2(n+1)(\ln n + \gamma - 1.5) + O(1) \\
 \therefore T(n) &= O(n \log n)
 \end{aligned}$$

Lec 05 基于归纳的算法设计 (核心章节)

算法设计的基本思想----从简单到复杂

将复杂的待求解问题变成简单问题有两种途径

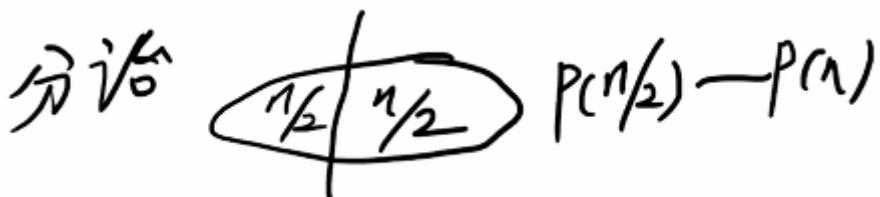
- 对问题进行横向分解，将原问题分解为一系列简单的互不相同的子问题，即模块化设计。
- 将问题进行纵向分解，即降低问题的规模。

纵向分解

原理

- 解决问题的一个小规模事例是可能的(基础事例)
- 每一个问题的解答都可以由更小规模问题的解答构造出来(归纳步骤)。
- 关键：如何简化问题。

1. 归纳/尾递归 $P(n) \rightarrow P(n-1)$
2. 增强假设 $P(n) \rightarrow P(n)Q(n)$
3. 分治法 $P(\frac{n}{2}) \rightarrow P(n)$
4. 动态归户撒



多项式求值

问题：给定一串实数 $a_n, a_{n-1}, \dots, a_1, a_0$ ，和一个实数 x ，计算多项式 $P_n(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ 的值。

归纳假设：已知如何在给定 a_{n-1}, \dots, a_1, a_0 和点 x 的情况下求解多项式（即已知如何求解 $P_{n-1}(x)$ ）。

$$P_n(x) = P_{n-1}(x) + a_nx^n$$

- 需要 $n(n+1)/2$ 次乘法和 n 次加法运算。

观察：有许多冗余的计算，即 x 的幂被到处计算。

更强的归纳假设：已知如何计算多项式 $P_{n-1}(x)$ 的值，也知道如何计算 x^{n-1} 。

计算 x^n 仅需要一次乘法，然后再用一次乘法得到 a_nx^n ，最后用一次加法完成计算，总共需要 $2n$ 次乘法和 n 次加法。

归纳假设（翻转了顺序的）：已知如何计算 $P'_{n-1}(x) = a_nx^{n-1} + a_{n-1}x^{n-2} + \dots + a_1$ 。

$P_n(x) = xP'_{n-1}(x) + a_0$ 。所以，从 $P'_{n-1}(x)$ 计算 $P_n(x)$ 仅需要一次乘法和一次加法。

- 该算法仅需要 n 次乘法和 n 次加法，以及一个额外的存储空间。

窍门是很少见的从左到右地考虑问题的输入，而不是直觉上的从右到左。另一个常见的可能是对比自上而下与自下而上（当包含一个树结构时）。

最大导出子图

令 $G=(V, E)$ 是一个无向图。一个 G 的导出子图是一个图 $H=(U, F)$ ，满足 $U \subseteq V$ 且 U 中两顶点若在 E 中有边则该边也包含在 F 中。

问题：给定一个无向图 $G=(V, E)$ 和一个整数 k ，试找到 G 的一个最大规模的导出子图 $H=(U, F)$ ，其中 H 中所有顶点的度 $\geq k$ ，或者说明不存在这样的子图。

解决问题的一个直接方法是把度 $< k$ 的顶点删除。当顶点连同它们连接的边一起被删除后，其它顶点的度也可能会减少。当一个顶点的度变成 $< k$ 后它也会被删除。但是，删除的次序并不清楚。我们应该首先删除所有度 $< k$ 的顶点，然后再处理度减少了的顶点呢？还是应该先删除一个度 $< k$ 的顶点，然后继续处理剩下受影响的顶点？⁶

- 任何度 $< k$ 的顶点都可以被删除。**删除的次序并不重要**。这种删除是必须的，而删除后剩下的图必定是最大的

设 $n-1$ 时命题. 当 $|V|=n$.

1) $n \leq k$

2) $n = k+1$ 完全图

3) $n > k+1$

i). 所有点的度数 $\geq k$.

ii). $\exists v \quad d(v) < k$

$d(v) \leftarrow x$

$\therefore v$ 在 G 的任意导出子图中
度数都小于 k

$\therefore v$ 必然不能包含在符合命题要
求的子图中

\therefore 可以去掉 v 及其邻结点
剩 $n-1$ 个点 归纳假设

寻找一对一映射

问题：给定一个集合 A 和一个从 A 到自身的映射 f , 寻找一个元
素个数最多的子集 $S \subseteq A$, S 满足：

- (1) f 把 S 中的每一个元素映射到 S 中的另一元素(即, f 把 S 映射到
它自身),
- (2) S 中没有两个元素映射到相同的元素(即, f 在 S 上是一个一
对一函数)。

归纳假设：对于包含n-1个元素的集合，如何求解问题是已知的。

假定有一个包含n个元素的集合A，并且要寻找一个满足问题条件的子集。我们断言，任何没有被其它元素映射到的元素i，不可能属于S。否则，如果*i*∈S且S有k个元素，则这k个元素映射到至多k-1个元素上，从而这个映射不可能是一对一的。如果存在这样的一个i，则我们简单地把它从集合中删除。现在我们得到集合A'=A-{i}，其元素个数为n-1，由归纳假设，我们已知对A'如何求解。如果不存在这样一个i，则映射是一对一的，即为所求，结束。

```
//O(n)
//Algorithm Mapping(f,n)
//Input: f (an array of integers whose values are between 1 to n)
//Output: S (a subset of the integers from 1 to n, such that f is one-to-one on
S)
begin
    S:=A;      //A is the set of numbers from 1 to n
    for j:=1 to n do c[j]:=0;
    for j:=1 to n do increment c[f[j]];
    for j:=1 to n do
        if c[j]=0 then put j in Queue;
    while Queue is not empty
        remove i from the top of the queue;
        S:=S-{i};
        decrement c[f[i]];
        if c[f[i]]=0 then put f[i] in Queue
end
```

社会名流问题

在n个人中，一个被所有人知道但却不知道别人的人，被定义为社会名流。

最坏情况下可能需要问n(n-1)个问题（你认识某人吗？）

问题：给定一个 $n \times n$ 邻接矩阵，确定是否存在一个i，其满足在第i列所有项(除了第ii项)都为1，并且第i行所有项(除了第ii项)都为0。

考察n-1个人和n个人问题的不同。由归纳法，我们假定能够在n-1个人中找到社会名流。由于至多只有一个社会名流，所以有三种可能：(1) 社会名流在最初的n-1人中，(2) 社会名流是第n个人，(3) 没有社会名流。但仍有可能需要n(n-1)次提问

“倒推”考虑问题。确定一个社会名流可能很难，但是确定某人不是社会名流可能会容易些。如果我们把某人排除在考虑之外，则问题规模从n减小到n-1。

算法如下：问A是否知道B，并根据答案删除A或者B。假定删除的是A。则由归纳法在剩下的n-1个人中找到一个社会名流。如果没有社会名流，算法就终止；否则，我们检测A是否知道此社会名流，而此社会名流是否不知道A。

算法被分为两个阶段：

1. 通过消除只留下一个候选者，
2. 检查这个候选者是否就是社会名流。

至多要询问 $3(n-1)$ 个问题：

第一阶段的 $n-1$ 个问题用于消除 $n-1$ 个人，

而为了验证候选者就是社会名流至多要 $2(n-1)$ 个问题。

```

Algorithm Celebrity(Know)
Input: Know (an n*n Boolean matrix)
Output: celebrity
begin
    i=1;
    j=2;
    next=3;
    {in the first phase we eliminate all but one candidate}
    while next<=n+1 do
        if Know[i,j] then i=next
        else j=next;
        next=next+1;
    if i=n+1 then candidate=j
    else candidate=i
    {now we check that the candidate is indeed the celebrity}
    wrong:=false;
    k=1;
    Know[candidate,candidate]=false;
    while not wrong and k<=n do
        if Know[candidate,k] then wrong=true;
        if not Know[k,candidate] then
            if candidate!=k then wrong=true;
        k=k+1;
    if not wrong then celebrity=candidate
    else celebrity=0
end

```

轮廓问题——分治法

问题：给定城市里几座矩形建筑的外形和位置，画出这些建筑的(两维)轮廓，并消去隐藏线。

建筑 B_i 通过三元组 (L_i, H_i, R_i) 来表示。 L_i 和 R_i 分别表示建筑的左右x坐标，而 H_i 表示建筑的高度。一个轮廓是一列x坐标以及与它们相连的高度，按照从左到右排列。

直接方法：每次加一个建筑，求出新的轮廓线，但总的步数为 $O(n^2)$

$$\underline{T(k) = T(k-1) + O(k) = O(k^2)}$$

发现合并两个轮廓线步数也是 $O(n^2)$

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

分治法

- 分治算法后的关键思想是：在最坏情况下，把一栋建筑与已有轮廓合并只需要线性的时间，并且两个不同轮廓合并也需要线性的时间。
- 使用类似于把一栋建筑与已有轮廓合并的算法，就能够把两个轮廓合并。我们从左到右同时扫描两个轮廓，匹配x坐标，并在需要时调整高度。这个合并可以在线性时间内完成，因此在最坏情况下完整的算法运行时间是 $O(n \log n)$ 。
- 分治法通过减小问题规模的途径来降低求解的难度。
- 但是分治法通常将问题拆分成多个相互独立但结构与原问题一致的小规模问题，然后将求解得到的小规模问题的解组合起来得到原问题的解。

分治法框架

```

divide-and-conquer(P)
1. if |P|<=n0 then solve(P)
2. 将P分拆为小规模的子问题P1, P2, ..., Pk
3. for i=1 to k
4.     yi=divide-and-conquer(pi)
5. end for
6. return merge(y1,...,yk)

```

从这一算法流程可以看到，分治法可以分为三个部分：

1. 分(divide): 将规模为n的原问题分拆为 $k \geq 1$ 个小问题，每个小问题的规模都严格小于n；
2. 治(conquer): 如果小问题的规模大于事先设定的阈值 n_0 ，则递归求解，否则直接求解小规模问题
3. 合(merge): 将k个子问题的解组合形成原问题的解

$$T(n) = div(n) + \sum_{i=1}^k T(n_i) + merge(n)$$

在二叉树中计算平衡因子——增强归纳假设

令T是一个根为r的二叉树。节点v的高度是v和树下方最远叶子的距离。节点v的平衡因子被定义成它的左子树的高度与右子树的高度的差

问题：给定一个n个节点的二叉树T，计算它的所有节点的平衡因子

归纳假设：我们已知如何计算节点数 $< n$ 的二叉树的全部节点的平衡因子。

然而，根的平衡因子，并不依赖于它的儿子的平衡因子，而是依赖于它们的高度，而高度很容易计算。

更强的归纳假设：已知如何计算节点数 $< n$ 的二叉树的全部节点的平衡因子和高度。

寻找最大连续子序列——增强归纳假设

问题：给定实数序列 x_1, x_2, \dots, x_n (不需要是正数)，寻找连续子序列 x_i, x_{i+1}, \dots, x_j ，使得其数值之和在所有的连续子序列数值之和中为最大。称这个子序列为最大子序列。

归纳假设：已知如何找到规模 $< n$ 的序列的最大子序列。

考虑规模 $n > 1$ 的序列 $S = (x_1, x_2, \dots, x_n)$ 。由归纳假设已知如何在 $S' = (x_1, x_2, \dots, x_{n-1})$ 中找到最大子序列。如果其最大子序列为空，则 S' 中所有的数值为负数，我们仅需考察 x_n 。假设通过归纳法在 S' 中找到的最大子序列是 $S'_M = (x_i, x_{i+1}, \dots, x_j)$, $1 \leq i \leq j \leq n-1$ 。如果 $j = n-1$ (即最大子序列是 S' 后缀)，则容易把这个解扩展到 S 中：若 x_n 是正数，则把它加到 S'_M 中；否则， S'_M 仍是最大子序列。如果 $j < n-1$ ，则或者 S'_M 仍是最大，或者存在另一个子序列，它在 S' 中不是最大，但在增加了 x_n 的 S 中是最大者。

19

更强的归纳假设：已知如何找到规模 $< n$ 的序列的最大子序列，以及作为后缀的最大子序列。

如果知道这两个子序列，算法就明确了。我们把 x_n 加到最大后缀中，如果它的和大于原来的最大子序列，则得到一个新的最大子序列(同样也是一个后缀)，否则，保留以前的最大子序列。但求解过程还没有完全结束，还需要寻找新的最大后缀子序列。我们不能总是简单地把 x_n 加到以前的最大后缀中，有可能以 x_n 结束的最大后缀的和是负数，在此情况下，把空集作为最大后缀。

```
Algorithm Maximum_Consecutive_Subsequence(X,n)
Input: X (an array of size n)
Output: Global_Max (the sum of the maximum subsequence)
begin
    Global_Max:=0;
    Suffix_Max:=0;
    for i:=1 to n do
        if x[i]+Suffix_Max>Global_max then
            Suffix_Max:=Suffix_Max+x[i];
            Global_Max:=Suffix_Max
        else if x[i]+Suffix_Max>0 then
            Suffix_Max:=x[i]+Suffix_Max
        else Suffix_Max:=0
    end
```

增强归纳假设

当试图用归纳方式证明时，我们经常遇到以下情节：用P来表示定理，归纳假设可以用 $P(<n)$ 表示，而证明必须推导出 $P(n)$ ，即 $P(<n) \Rightarrow P(n)$ 。在许多情况下，我们可以增加另一个假设，称之为Q，从而使证明变得容易，即证明 $[P \text{ and } Q](<n) \Rightarrow P(n)$ 比证明 $P(<n) \Rightarrow P(n)$ 容易

在使用这个技巧时，人们最易犯的错误是增加的额外假设本身必须也有相应的证明。换句话说，当他们证明 $[P \text{ and } Q](<n) \Rightarrow P(n)$ 时，忘记了Q是假定的。

至关重要的是要精确地按照归纳假设进行问题求解。

背包问题——动态规划

问题：给定一个整数K和n个不同大小的物品，第*i*个物品的大小为整数 k_i ，寻找一个物品的子集，它们的大小之和正好为K，或者确定不存在这样的子集。

用 $P(n, K)$ 表示该问题，其中n表示物品的数目而K表示背包的大小。关注判定问题。

归纳假设(最初的设想)：已知如何求解 $P(n-1, K)$ 。但如果设对于 $P(n-1, K)$ 不存在解。我们可以使用这个否定的结论吗？

归纳假设(第二次设想)：我们已知如何求解 $P(n-1, k)$ ，其中 $0 \leq k \leq K$ 。但这个算法可能是低效率的，我们把一个规模为n的问题归约到了两个规模为n-1的子问题！

- 关键：全部的可能问题数目不是很大，在许多次得到的是同样的问题。可以在求解中记住已有的解答，从而对相同的问题不作第二次求解。
- 方法：把增强的归纳假设与强归纳(它不仅利用n-1时的解，而是利用所有较小规模情形时的解)结合起来。
- 动态规划的本质是把所有前面已知的结果建成一个大表格。这个表格是被迭代构造的。每一项是由矩阵中它上面的其它项结合计算得出，或者是由它左边的项结合计算得出。主要的问题是用最高效的方式来组织矩阵的构造。

```
Algorithm Knapsack(S, K)
Input: S (an array of size n storing the sizes of the items), K (the size of the knapsack)
Output: P (a 2D-array such that P[i,k].exist=true if there exists a solution with the first i elements and a knapsack of size k, and P[i,k].belong=true if the ith element belongs to that solution)
begin
    P[0,0].exist:=true;
    for k:=1 to K do
        P[0,k].exist:=false;
    for i:=1 to n do
        for k:=0 to K do
            P[i,k].exist:=false;
            if P[i-1,k].exist then
```

```

P[i,k].exist:=true;
P[i,k].belong:=false;
else if k-s[i]>=0 then
    if P[i-1,k-s[i]].exist then
        P[i,k].exist:=true;
        P[i,k].belong:=true;
end

```

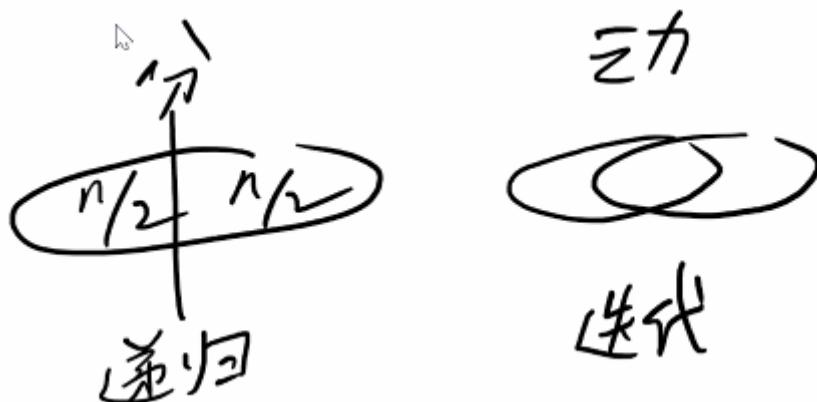
复杂性：在表格中有 nK 个项，每一项由其他两项在常数式时间内计算所得。因此，总共的运行时间是 $O(nK)$ 。

是一个指数时间的算法，有 n 和 k 两个参数都会影响运行时间， n 是物品的个数， k 是背包容量大小，bit数是 $\log k$ ， $k = 2^{\log k}$

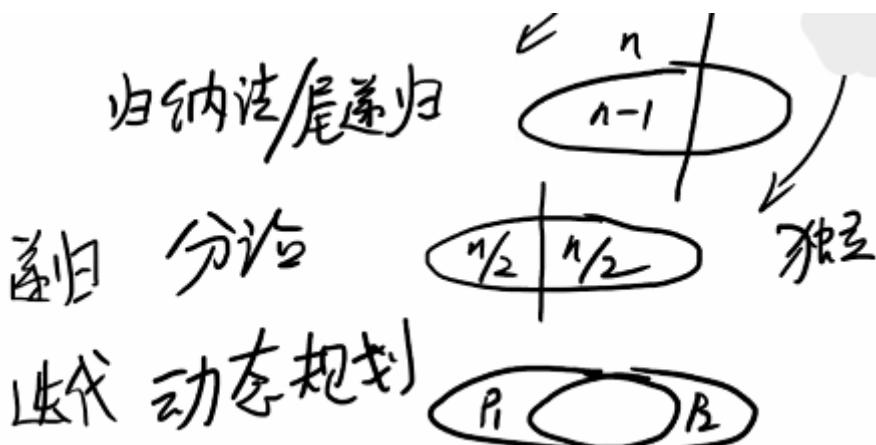
动态规划

如果算法在求解问题的过程中需要反复计算某些子问题，则可以从小规模问题开始求解，并将其解存储起来，在求解大规模问题时，如果需要某些子问题的解，则利用存储内容直接获得其解，避免重复计算。

动态规划与分治法差异



小结



常见错误

与已经讨论的归纳证明的常见错误类似。

例如，忘记了基础情形是比较常见的。在递归过程中，基础情形对于终止递归是必需的。

另一常见错误是，把对于 n 的解扩展到问题对于 $n+1$ 时的一个特定实例的解，而不是对于任意实例。

无意识的改变假设是另一个常见错误。

归纳原理

通过把问题的一个实例归约到一个或多个较小规模的实例，可以使用归纳原理来设计算法。如果归约总是能实现，并且基础情形可以被解决，则算法可通过归纳进行设计。因此，主要的思想是如何归约问题，而不是直接对问题求解。

把问题规模减小的一种最容易的方法是去除问题中的某些元素。这个技术应该是处理问题首要手段，可以有许多不同的方式，除了简单地去除元素外，把两个元素合并为一个也是可能的，或者找到一个在特定(容易)情况下可以处理的元素，或者引入一个新元素来取代原来的两个或几个元素。

可以用多种方式来归约问题的规模。然而，不是所有的归约都有同样的效率，因此要考虑所有归约的可能性，特别是考虑不同的归纳次序。

减小问题规模的一个最有效的方法是把它分成两个(或多个)相等规模的部分。如果问题可以被分割，则分治法非常有效，其中子问题的输出可以容易地生成全问题的输出。

由于归约只能改变问题的规模，并不改变问题本身，所以应该寻找尽可能独立的小规模的子问题。

有一种方法来克服归约问题必须与原始问题一致的局限：改变问题的描述。这是一个经常使用的非常重要的方法，有时，它比削弱假设要好，可获得一个较弱的算法并作为完整算法中的一个步骤来使用。

这些技术可以同时一起使用，或者作不同的组合。

归纳法（尾递归），分治法，动态规划

Ch 06 序列和集合的算法

序列与集合

序列须考虑元素的顺序，而集合则不必

集合中的元素不能重复出现，而序列则没有这样的限制

序列或集合中的元素取自全序集合（如整数集合、实数集合等），由此任意元素之间可以比较大小

二分搜索

二分搜索

问题：给定实数序列 x_1, x_2, \dots, x_n ，满足 $x_1 \leq x_2 \leq \dots \leq x_n$ 。对某个实数 z ，试确定 z 是否在该序列中出现。若 z 出现，试确定下标 i 使 $x_i = z$ 。

为简单起见，这里只查找一个满足 $x_i = z$ 的下标 i 。

基本思想：通过一次查询检验中间数把搜索空间（大致）减半

算法复杂度：算法每进行一次比较，搜索的范围就减半，因此，在 n 个元素的序列中查找某个数值需要比较 $O(\log n)$ 次。

循环序列中二分搜索

例如3 4 5 1 2

如果在序列 x_1, x_2, \dots, x_n 中，存在某个 i 使 x_i 是序列中的最小者，且序列 $x_i, x_{i+1}, \dots, x_n, x_1, \dots, x_{i-1}$ 是递增的，则称序列 x_1, x_2, \dots, x_n 是循环序列。

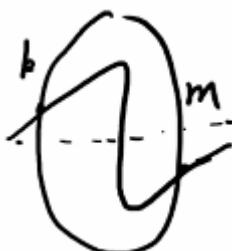
问题：对某个已知的循环排序的链表，找出其中最小元素的位置。为简单起见，假设该位置是唯一的。

思想：利用二分搜索，通过一次比较将序列长度减半。在序列中任取两个元素 x_k 和 x_m ，其中 $k < m$ 。若 $x_k < x_m$ ， i 就不可能落在 $k < i \leq m$ 中，因为 x_i 是序列中的最小数（注意，这里不能排除 x_k ）。另一方面，若 $x_k > x_m$ ， i 就一定落在 $k < i \leq m$ 中，因为在这个区间内元素的顺序发生了跳变。这样一来，比较一次元素大小就能排除许多元素。通过选择恰当的 k 和 m 就能在 $O(\log n)$ 次比较后找到 i 。

第一种



第二种



```
Algorithm Cyclic_Binary_Search(X,n)
Input: X, a cyclically sorted array in the range 1 to n of distinct elements
Output: Position (the index of the minimal element)

begin
    Position:=Cyclic_Find(1,n);
end

function Cyclic_Find(Left,Right):integer;
begin
    if Left=Right then Cyclic_Find:=Left
    else
        Middle:=向下取整(Left+Right)/2;
        if X[Middle]<X[Right] then
            Cyclic_Find:=Cyclic_Find(Left,Middle)
        else
            Cyclic_Find:=Cyclic_Find(Middle+1,Right)
end
```

二分搜索特殊下标

问题：给定一个由不同整数 a_1, a_2, \dots, a_n 按升序排列而成的序列，试确定是否存在某个下标*i*使得 $a_i=i$ 。

思想：考察 $a_{n/2}$ 值（假设n是偶数），如果它的值恰好是n/2，则问题得解。否则的话，若该值小于n/2，则由所有元素各不相同可知， $a_{n/2-1}$ 的值小于n/2-1，依次类推，前半段序列中的元素均不满足性质，可对后半段继续搜索。若该值大于n/2，则同样可以得到相似的结论。

```
Algorithm Special_Binary_Search(x,n)
Input: X, a sorted array in the range 1 to n of distinct integers
Output: Position (the index of satisfying A[position]=Position, or 0 if no such
index exists)

begin
    Position:=Special_Find(1,n);
end

function Special_Find(Left,Right):integer;
begin
    if Left=Right then
        if A[Left]=Left then Special_Find:=Left
        else Special_Find:=0
    else
        Middle:=向下取整(Left+Right)/2;
        if X[Middle]<Middle then
            Special_Find:= Special_Find(Middle+1,Right)
        else
            Special_Find:= Special_Find(Left,Middle)
    end
end
```

二分搜索长度未知的序列

考虑一般的搜索问题，但序列长度未知。

此时不能把搜索空间减半，因为序列边界并不清楚。

查找大于等于z的某个 x_i 。如果找到了这个 x_i ，就可以在下标为1到*i*的范围内进行二分搜索了。

首先将z同 x_1 比较。若 $z \leq x_1$ ，则z只可能等于 x_1 。借助归纳法，假设对某个 $j \geq 1$ 有 $z > x_j$ ，将z同 x_{2j} 比较，搜索空间加倍。若 $z \leq x_{2j}$ ，可推得 $x_j < z \leq x_{2j}$ ，然后再通过 $O(\log j)$ 次比较就能找到z。总的来看，如果*i*是使 $z \leq x_i$ 的最小下标，那么通过 $O(\log i)$ 次比较就能得到某个 x_j ，使得 $z \leq x_j$ ，然后再通过 $O(\log i)$ 次比较最终得到*i*。

重叠子序列问题

子序列不要求连续，子串必须连续

令A和B是有限字母表上的字符序列， $A=a_1a_2\dots a_n$, $B=b_1b_2\dots b_m$, 其中 $m \leq n$ 。若存在下标 $i_1 < i_2 < \dots < i_m$, 对所有的 j , $1 \leq j \leq m$, 有 $b_j = a_{i_j}$, 则称B是A的子序列。即如果能把B中的字符按序嵌入A中，嵌入的位置不必连续，则B是A的子序列。

从头开始扫描A直到 b_1 第一次出现（如果存在的話），继续扫描直到 b_2 出现，以此类推。由于算法中包含一次对A和B的线性扫描，因而运行时间为 $O(m+n)$ 。

对已知的序列B，定义 B^i 为B的每个字符连续出现i次所形成的序列。

问题：给定两个序列A、B，试确定i的最大值，使得 B^i 是A的子序列。

对每个已知的i，构造序列 B^i 是轻而易举的。

对任意给定的i，也能够判定 B^i 是否为A的子序列。

如果 B^j 是A的子序列，那么对 $1 \leq i \leq j$, B^i 也是A的子序列。

i的最大值不能超过 n/m , 否则序列 B^i 的长度就会超过A。

首先令 $i=\lceil n/m \rceil/2$, 检验 B^i 是否为A的子序列。然后根据结果，

如果它是A的子序列，则 i 介于 $\lceil n/m \rceil/2$ 和 $\lceil n/m \rceil$ 间，否则就考虑 1 到 $\lceil n/m \rceil/2$ 的情况。这样共需进行 $\lceil \log(n/m) \rceil$ 次检验才能确定i的最大值，耗费的总时间为 $O((n+m)\log(n/m))=O(n\log(n/m))$ 。

启示

每当要查找满足某个特定性质的最大i，首先需要一个能判断i是否满足该性质的算法。

如果i有上界：若i满足该性质，对 $1 \leq j \leq i$, j 也满足，那么接下来就可以借助二分搜索来解决。

若i的上界未知：可以采取加倍法。即从 $i=1$ 开始，对i的值加倍直至找到合适的范围。

方程求解

设欲解方程 $f(x)=0$ ，其中 f 是可计算的连续函数。现已知 x 落在区间 $[a,b]$ 中（ $a \leq x \leq b$ ），且 $f(a) \cdot f(b) < 0$ 。求该方程在指定精度下的解。

由函数的连续性可知区间 $[a,b]$ 中一定有解。于是就可以利用二分搜索的变形二分法来求解。

取 $x_1 = (a+b)/2$ ，若 $f(x_1) = 0$ （在给定的精度下）， x_1 就是方程的解。
否则解一定在 $[a,b]$ 的半区间 $[a, x_1]$ 或 $[x_1, b]$ 中，取区间端点函数值一正一负的那个区间继续进行上述过程，直到求得给定精度下的解。在 k 步之后，含解区间的长度为 $(b-a)/2^k$ 。

内插搜索

二分搜索总是把搜索空间减半，从而保证其性能是对数级的。而当某一步所得到的值十分接近目标值时，在该值的邻域内搜索似乎比盲目地减半搜索空间更加合理。

内插搜索的性能不仅取决于序列的长度，还和序列自身的特点有关。对有些序列，内插搜索不得不一一检查其中的每个数，而对于数据分布相对均匀的序列，内插搜索就十分高效。

内插搜索的平均比较次数是 $O(\log \log n)$ 。尽管内插搜索比二分搜索在性能上有指数阶的进步，但在实际性能上并无多大改善：1)除非 n 很大，否则 $\log n$ 已经足够小以至于其对数并不比其本身小多少；2)内插搜索需要更复杂的数值运算。

```
Algorithm Interpolation_Search(x,n,z)
Input: x (a sorted array in the range 1 to n), z (the search key)
Output: Position (an index i such that x[i]=z, or 0 if no such index exist)
begin
    if z<x[1] or z>x[n] then position:=0
    else Position:=Int_Find(z,1,n)
end

function Int_Find(z,Left,Right):integer;
begin
    if x[Left]=z then Int_Find:=Left
    else if Left=Right or x[Left]=x[Right] then
        Int_Find:=0
    else
        Next_Guess:=向上取整Left+(z-x[Left])(Right-Left)/(x[Right]-x[Left]);
        if z<x[Next_Guess] then
            Int_Find:=Int_Find(z,Left,Next_Guess-1)
        else
            Int_Find:=Int_Find(z,Next_Guess,Right)
end
```

排序

问题：对已知的n个数 x_1, x_2, \dots, x_n 从小到大排序，也就是要找到由n个不同的下标所组成的序列 $1 \leq i_1 < i_2 < \dots < i_m \leq n$ ，使得 $x_{i_1} \leq x_{i_2} \leq \dots \leq x_{i_m}$

为简单起见，假设数据均不相同。

如果某个排序算法除存储原始数据的数组之外不再借用额外的工作空间，则称之为原地置换排序算法。

插入排序

假设知道应该如何对n-1个数进行排序。现在给定n个数，先对前n-1个数排序，然后通过扫描已排完序的n-1个数来确定第n个数应该插入的位置。

最坏情况下，第n个数不得不跟前n-1个数挨个比较后才能找到合适的位置。因此对n个数排序可能就要作高达 $1+2+\dots+n-1 = (n-1)(n-2)/2 = O(n^2)$ 次比较。

插入第n个数时还需要移动其他的数。最坏情况下第n步就有可能移动n-1个数，因而总的移动次数同样也是 $O(n^2)$ 。

可以把数存储在数组中，然后对n-1个已排序的数用二分搜索来确定插入第n个数的正确位置，从而改进算法的性能。这样每插入一个数搜索过程就进行 $O(\log n)$ 次比较，总体比较次数降为 $O(n \log n)$ 。但数据移动次数没有减少，所得算法仍然是 $O(n^2)$ 的。

18

选择排序

把最大数选作第n个数。选最大数主要是因为我们知道它应该被放在哪个位置上。

首先选出最大的数，然后放在其应处的位置上（只需和该位置上原有的数交换即可），然后递归地对剩余的数进行排序。

选择排序移动n-1个数据（这里指数据交换），而插入排序在最坏情况下的移动次数为 $O(n^2)$ 。

由于选出最大数需n-1次比较，总的比较次数仍为 $O(n^2)$ ，而借助二分搜索的插入排序仅需比较 $O(n \log n)$ 次。

冒泡法也是一种选择排序，选出最大数的做法不同

归并排序

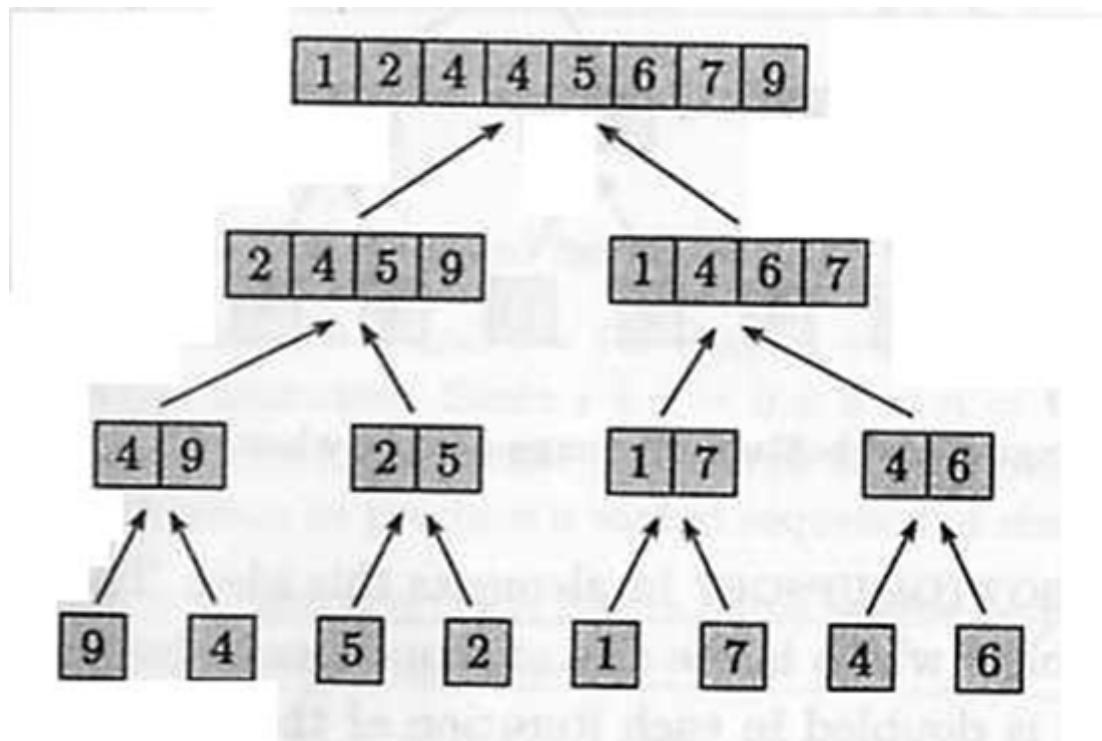
插入排序的推广

插入排序 → 归并排序

$$\begin{aligned}
 & n-1 \rightarrow + n_1 \\
 & n-1 + 1 \rightarrow n \\
 & T(n) = T(n-1) + O(n) = O(n^2) \\
 & n/2 \rightarrow + n/2 \\
 & n \rightarrow \\
 & T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)
 \end{aligned}$$

记 a_1, a_2, \dots, a_n 为第一个集合中的元素， b_1, b_2, \dots, b_n 为第二个集合中的元素，两集合中的元素均以升序排列。扫描第一个序列，直到找到插入 b_1 的位置，然后插入 b_1 ；接着在插入处继续扫描，直到找到插入 b_2 的位置，如是依次插入余下的数。
由于 b 序列是有序的，因此不必回溯查找。 最坏情况下的比较次数是两个序列长度之和。因归并后的数是按序排列的，因此可在归并过程中将其复制到一个临时数组，这样每个数恰被复制一次。于是归并两个长度分别为 n 和 m 的有序序列共需 $O(n+m)$ 次比较和移动。

归并排序算法：首先将序列分成相等的两部分，然后将每个部分分别递归排序，最后把这两个有序部分按上述过程归并为一个有序序列。



Algorithm MERGE

Input: An array $A[1..m]$ of elements and three indices p , q and r , with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order
Output: $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$

```
s←p, t←q+1, k←p
while s≤q and t≤r
    if A[s]≤A[t] then
        B[k]←A[s]      {B[p..r] is an auxiliary array}
        s←s+1
    else
        B[k]←A[t]
        t←t+1
    end if
    k←k+1
endwhile
if s=q+1 then B[k..r]←A[t..r]
else B[k..r]←A[s..q]
endif
A[p..r]←B[p..r]
```

Algorithm MERGESORT

Input: An array $A[1..n]$ of n elements

Output: $A[1..n]$ sorted in nondecreasing order

```
t←1
while t<n
    s←t, t←2s, i←0
    while i+t≤n
        MERGE(A, i+1, i+s, i+t)
        i←i+t
    end while
    if i+s<n then MERGE(A, i+1, i+s, n)
endwhile
```

复杂度

最坏情况下归并排序所需比较的次数 $T(n)=O(n\log n)$ ($T(2n)=2T(n)+O(n)$, $T(2)=1$)

数据移动的次数同样是 $O(n \log n)$, 比选择排序的 $O(n)$ 要多。

缺点

首先, 归并排序不易实现; 第二, 归并过程中需要额外的空间来存储归并后的序列, 因此它不是一种原地置换排序算法。每次归并两个短序列时都要复制所有的数, 因而算法效率有所下降。

快速排序

归并排序的症结在于额外存储空间的使用，由于合并过程是不确定的，因此每个数在序列中的最终位置是不可预知的。那么究竟能不能采取一种分治思想使得每个数的最终位置是确定的呢？快速排序思想正是将“分”作为重点，而不是在“治”。

假设知道某个数x满足：序列中一半的数大于等于x，另一半小于x。根据和x的大小关系，可以将序列一分为二。这样的分割需要n-1次比较。序列分割完之后，接着将每个子序列递归排序。最后的合并自然而成，因为两个子序列本来就存储在数组中。整个排序过程不需要任何额外的空间。

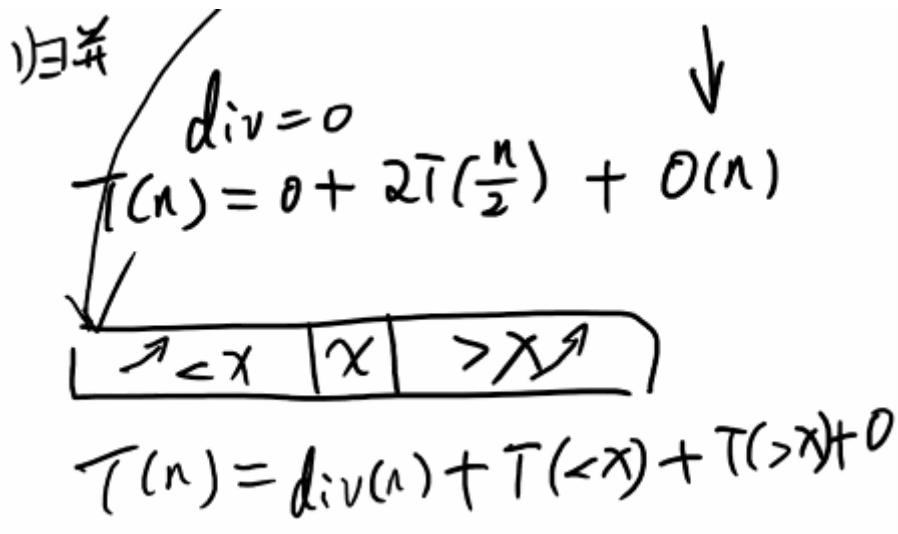
上述排序过程基于一个假设，x的值是预知的，不过算法在x未知时依然有效。把分割步骤中的这个数称为主元。把序列分为两部分，一部分的数大于主元，另一部分小于等于主元。

用两个指针L和R分别指向数组的左右两侧，然后指针沿数组项相向而行。

归纳假设：在算法的第k步，对任意i, i<L, 都有主元pivot>xi；对任意j, j>R, 都有主元pivot<xj。

如何选择一个合适的主元：分治算法在各个部分大小相等时性能最好，即主元越靠近序列的中点，算法的运行效率就越高。可以采用序列的中数，但如果序列本身是随机排列的，那不妨就把第一个数选作主元。

归并和快排



归并关注merge，快排关注divide

```
Algorithm Partition(X,Left,Right)
Input: X (an array), Left (the left boundary of the array), Right (the right boundary)
Output: X and Middle such that X[i]≤X[Middle] for all i≤Middle and X[j]>X[Middle] for all j>Middle
Begin
    pivot:=X[Left];
    L:=Left; R:=Right;
    while L<R do
        while X[L]≤pivot and L≤Right do L:=L+1;
        while X[R]>pivot and R≥Left do R:=R-1;
        if L<R then exchange X[L] with X[R];
    Middle:=R;
    exchange X[Left] with X[Middle];
end
```

```

Algorithm QUICKSORT(X,n)
Input: X (an array in the range 1 to n)
Output: X (the array in sorted order)
begin
    Q_Sort(1,n)

Procedure Q_Sort(Left,Right)
Begin
    if Left<Right then
        Partition(X,Left,Right);
        Q_Sort(Left,Middle-1);
        Q_Sort(Middle+1,Right);
end

```

快排时间复杂度

平均时间

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

$$= (n-1) + \frac{2}{n} \sum_{i=0}^{n-1} T(i) = \underline{\underline{O(n \log n)}}$$

最坏情况 $O(n^2)$

堆排序

堆用隐式表示，即数组A[1..n]中存储着所有的数，数组与树的对应关系如下：树的根存在A[1]中，任意节点A[i]的子节点存在A[2i]和A[2i+1]中，每个节点的值都大于等于其子节点的值。

原理：输入是数组A[1..n]。首先，数组中的元素重新排列成堆。若A是堆，则A[1]是数组的最大元素，交换A[1]和A[n]使A[n]中存储正确的那个元素。然后考察数组A[1..n-1]，同样地，将这个数组中的所有元素排列成堆（其实只需考虑新的A[1]），交换A[1]和A[n-1]，接着继续考察A[1..n-2]。总的来说，堆排序包括一次初始的建堆过程，n-1次元素交换和重排成堆。

堆排序最坏情况下的运行时间是 $O(n \log n)$ ，是一个原地置换排序算法。

```

Algorithm HEAPSORT
Input: An array A[1..n] of n elements
Output: Array A sorted in nondecreasing order
1. MAKEHEAP(A)
2. for j<-n downto 2
3.     interchange A[1] and A[j]
4.     SIFT-DOWN(A[1..j-1],1)
5. end for

```

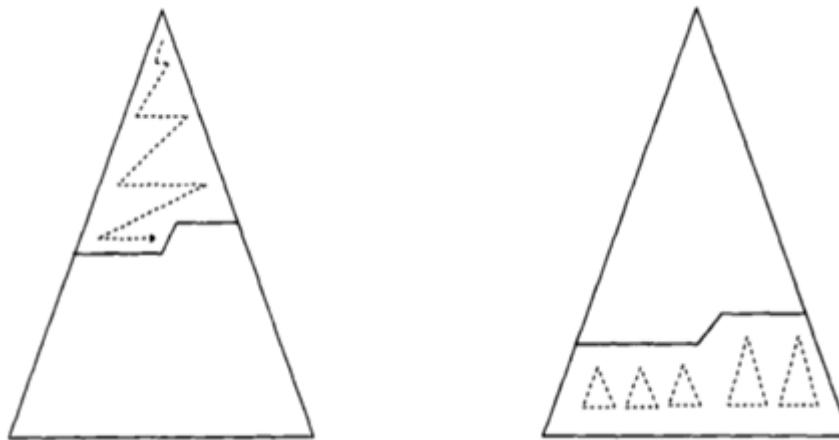
```

Procedure SIFT-DOWN
Input: An array H[1..n] and an index i between 1 and n
Output: H[i] is percolated down, if necessary, so that it is not smaller than
its children
done<-false
if 2i>n then exit {node i is a leaf}
repeat
    i<-2i
    if i+1<=n and key(H[i+1])>key(H[i]) then i<-i+1
    if key(H[向下取整i/2])<key(H[i]) then interchange H[i] and H[向下取整i/2]
    else done=true
    endif
until 2i>n or done

```

建堆

自顶向下法和自底向上是两个很自然的建堆方法，它们分别对应着两种不同的对数组的扫描方式：自左向右或自右向左。



自顶向下

先考虑从左到右对数组进行扫描（对应自顶向下建堆法）。

归纳假设(自顶向下): 数组A[1..i]是堆。

归纳基础是显见的，因为A[1]本身一定是堆。算法的主要部分

在于如何把A[i+1]放到堆A[1..i]中去，也就是如何在堆中插入A[i+1]。将A[i+1]与其父节点比较，然后不断交换节点的位置，直到新的父节点大与其自身。最坏情况下的比较次数是 $\lfloor \log(i+1) \rfloor$ 。

自底向上

再考虑从右到左对数组进行扫描（对应自底向上建堆法）。我们想要说明数组 $A[i+1..n]$ 是堆，然后如何在堆中插入 $A[i]$ 。可是， $A[i+1..n]$ 并不代表某个堆，它对应着数个堆。

归纳假设(自底向上): $A[i+1..n]$ 代表的所有树都满足堆的性质。 $A[n]$ 本身是堆，归纳基础成立。数组 $A[\lfloor n/2 \rfloor + 1..n]$ 表示树上的叶节点，对应着 $A[\lfloor n/2 \rfloor + 1..n]$ 的树都是单节点树，自然满足堆的性质。因此只需从 $\lfloor n/2 \rfloor$ 开始归纳。

考察 $A[i]$ ，它最多只能有两个子节点($A[2i]$ 和 $A[2i+1]$)，且都是其他某个堆的根节点。 $A[i]$ 先同它最大的子节点比较，如果有必要就和那个最大的子节点交换。沿着树自上而下不断交换直到原先的 $A[i]$ 值大于其现在两个子节点的值。由于 $A[i]$ 的高度是 $\lfloor \log(n/i) \rfloor$ ，最坏情况下的比较次数是 $2\lfloor \log(n/i) \rfloor$ 。

32

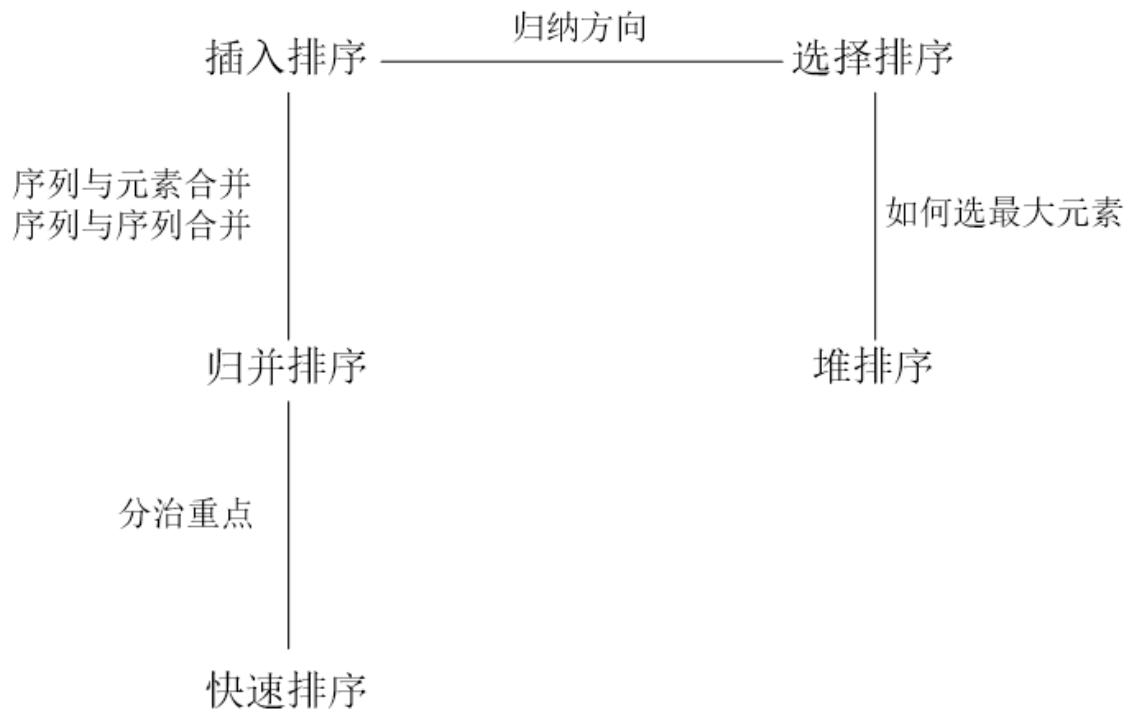
时间复杂度

自顶向下: 第 i 步最多需要 $\lfloor \log i \rfloor$ 次比较，故运行时间为 $O(n \log n)$

自底向上: 每一步比较次数最多是相应节点高度的两倍，所以算法复杂度最多是树上所有节点高度和的两倍。首先考察满二叉树，记 $H(i)$ 是高度为 i 的满二叉树所有节点的高度之和，并把高为 i 的树看作两棵高为 $i-1$ 的树再加上一个根节点。于是 $H(i)=2H(i-1)+i$ ， $H(0)=0$ 。可以证明该递归关系式的解是 $H(i)=2^{i+1}-(i+2)$ 。因为高为 i 的满二叉树有 $2^{i+1}-1$ 个子节点，可推得满二叉树（即有 2^k-1 个节点的堆）自底向上建堆的算法复杂度是 $O(n)$ 。那么为有 n 个节点的二叉树建堆， $2^k \leq n < 2^{k+1}-1$ ，其算法复杂度不会超过 $2^{k+1}-1$ 个节点的堆，故仍为 $O(n)$ 。

自底向上法比自顶向下法快主要是因为树底部的节点多于顶部，所以简化对底部节点的处理可得到更好的结果。

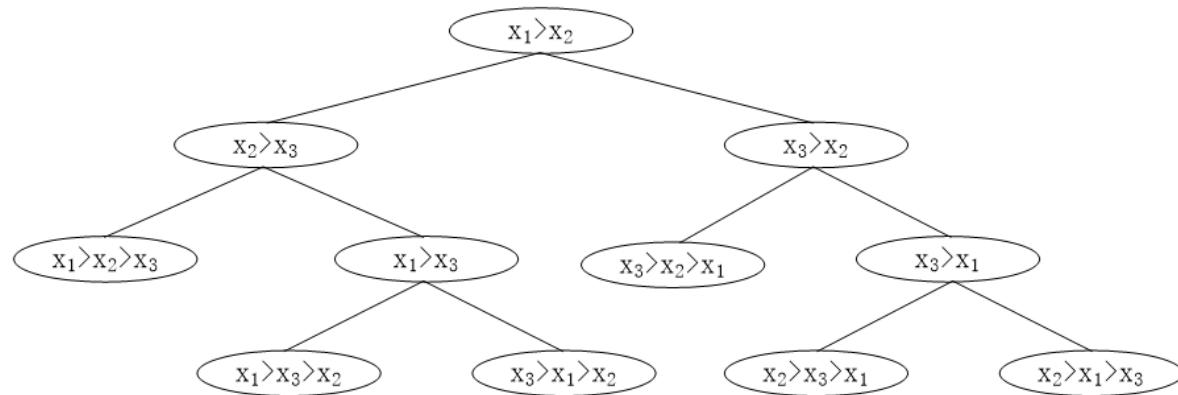
排序总结



决策树

决策树是由内部节点和叶子节点构成的二叉树。每个内部节点都是一个查询，查询结果有两种可能，分别由该节点发出的两条边表示，每个叶子节点都是一种可能的输出。

设算法的输入是序列 x_1, x_2, \dots, x_n 。计算过程从根节点开始，每个节点对输入进行查询，根据查询结果选择左、右分支，直到到达某个叶子节点，把该叶子节点的输出作为整个计算过程的输出。最坏情况下的运行时间是T的高度，相当于输入最多所需查询的次数。因此决策树事实上对应着算法，尽管它不能模拟所有的算法（比如说，决策树无法计算平方根），但是对于基于比较的算法而言，它的确是一个很合适的模型。由决策树所取得的下界意味着不存在比决策树更好、跟它的形式相同的算法。



定理6.1：任意一个排序算法的决策树高度为 $\Omega(n \log n)$ 。

此下界称为信息论下界，它和计算过程完全无关（请注意刚才甚至没有定义内部节点允许的查询种类），只和输出信息量有关。

问题下界的证明说明了基于比较的排序算法不可能比 $\Omega(n \log n)$ 快，不过可以通过利用关键字的特殊性质或者对关键字进行代数运算来加快排序速度。

如果问题的下界和某个算法的上界相等，那么这个下界就意味着算法不可能进一步优化。

不基于比较的排序 计数排序

前提：假设有n个取值范围为1至k的元素，当k=O(n)时，运行时间O(n)

基本思想：对每一个 x ，求出小于 x 的元素个数。有了这个信息就可把 x 直接放到其在最终输出数组中的位置上

```
Algorithm Countingsort(A, n, k)
Input: A (an array in the range 1 to n)
Output: B (the array in the sorted order)
begin
    for i:=1 to k
        C[i]=0
    for j:=1 to n
        C[A[j]]=C[A[j]]+1 {C[i]=值为i的元素个数}
    for i:=2 to k
        C[i]=C[i]+C[i-1] {C[i]=值小于等于i的元素个数}
    for j:=n downto 1
        B[C[A[j]]]=A[j];
        C[A[j]]=C[A[j]]-1;
end
```

桶排序

假设有 n 个取值范围为1至 m 的元素。为它们分配 m 个桶，然后对每个 i ，根据 x_i 的值，把 x_i 放到相应的桶中去。最后按序扫描每个桶，并把桶中的元素收集起来。

算法复杂度为 $O(m+n)$ 。一方面，若 $m=O(n)$ ，算法就演化成线性时间排序算法；另一方面，若 m 相对于 n 很大，则 $O(m)$ 也会很大，而且算法需要 $O(m)$ 的存储空间，这比 m 自身很大更让人头疼。

桶排序算法的一个拓展

假设有 n 个元素，均匀分布在区间 $[0,1]$ 上。

把区间 $[0,1]$ 划分成 n 个相同大小的区间/桶，然后根据 x_i 的值，把 x_i 放到相应的桶中去。由于输入元素是均匀分布的，一般不会有很多数落在一个桶中。先对每个桶中的数进行排序，最后按序扫描每个桶，把桶中的元素收集起来。

for $i:=1$ to n

将 $A[i]$ 插入到表 $B[\lfloor n \cdot A[i] \rfloor]$ 中

for $i:=0$ to $n-1$

用插入排序对表 $B[i]$ 进行排序

将表 $B[0], B[1], \dots, B[n-1]$ 按顺序合并

时间复杂性： $O(n)$

$\sum_{i=1}^n B[i]$ 中元素个数 随机变量
 $O(n_i^2)$
平均 $E(O(n_i^2)) = O(E(n_i^2))$

$$\begin{aligned}
 n_i &= j \quad \text{服从二项分布} \\
 b(j; n, p = 1/n) \\
 E n_i &= n \cdot p = 1 \quad \text{Var}(n_i) = np(1-p) \\
 &\quad = 1 - 1/n \\
 E n_i^2 &= (E n_i)^2 + \text{Var}(n_i) \\
 &= 1 - \frac{1}{n} + 1 = 2 - \frac{1}{n} = \textcircled{H}(1)
 \end{aligned}$$

$\sum_{i=0}^{n-1} E(O(n_i)^2) = O(n)$

基数排序

假设元素都是k位数的较大整数，每一位的取值范围为0到d-1。

归纳假设：知道该怎样用<k位的元素进行排序。以字典序排序的序列，最高位决定了元素排列的顺序，同其他位无关。另一方面，如果两个元素的最高位相同，则由归纳假设，在最后一步之前，它们已经按正确的顺序排列。因此，仅需保证其排列顺序不变。

关键：同一个桶内的元素排列顺序不变。

如已将对前 k-1 位排好序
底

在对第 k 位排序时

1). 第 k 位相同，保持前 k-1 位
时的次序

2). 第 k 位不同，按第 k 位大小
排序

例子

7467	6792	9134	9134	1239
1247	9134	1239	9187	1247
3275	3275	1247	1239	3275
6792	4675	7467	1247	4675
9187	7467	3275	3275	6792
9134	1247	4675	7467	7467
4675	9187	9187	4675	9134
1239	1239	6792	6792	9187

代码实现

```
Algorithm Straight_Radix(x,n,k)
Input: x (an array of n integers, each with k digits, in the range 1 to d)
Output: x (the array in sorted order)
begin
    Assume all elements are initially in a global queue GQ;
    for i:=1 to d do      {d is the number of possible digits}
        initialize queue Q[i] to be empty;
    for i:=k downto 1 do
        while GQ is not empty do
            pop x from GQ;
            f:=the ith digit of x;
            insert x into Q[f];
            for t:=1 to d do
                insert Q[t] into GQ;
        for i:= 1 to n do
            pop x[i] from GQ
    end
```

复杂度 $O(nk)$

秩

已知序列 $S = x_1, x_2, \dots, x_n$ 。若 x_i 是 S 中第 k 小的数，称 x_i 在 S 中的秩是 k 。通过排序，很容易就能知道每个数在序列中的秩。然而有些关于秩的问题不用排序也能解决。

最大和最小数

单独查找序列中的最大和最小数是很方便的。如果知道前 $n-1$ 个数构成的序列中的最大数，就只需将这个最大数同第 n 个数作比较，从而得到这 n 个数中的最大数（查找1个数构成的序列中的最大数是显然的）。以上过程中从第2个数开始，每个数比较一次，因此共需比较 $n-1$ 次。

问题：在已知的某个序列中查找最大和最小数。

直接法：分别查找最大数和最小数。这样一共要作 $2n-3$ 次比较： $n-1$ 次找到最大数， $n-2$ 次找到最小数（因为此时可以排除最大数）。

归纳法：设知道 $n-1$ 个数时问题的解，现在求 n 个数时的解。那么就只需要把第 n 个数与当前找到的最大、最小数分别比较。这样，每个数需两次比较，可以推得总共仍需 $2n-3$ 次。以不同顺序扫描序列中的数也不能改进其性能，这主要是由于序列中数的位置同问题是无关的。

扩展方案：已知 $n-2$ 个数的解，对 n 个数求解。考察 x_{n-1} 和 x_n ，设 $\text{MAX}(\text{min})$ 是前 $n-2$ 个数的最大（小）数，作三次比较便能得出新的最大最小数。首先比较 x_{n-1} 和 x_n ，然后将其中大数同 MAX 比较，小数同 min 比较。这样一来，总共只要大约 $3n/2$ 次比较。

$$\left\{ \begin{array}{l} C(n) = C(n-2) + 3 \\ \text{如果 } n = 2k \\ C(2) = 1 \\ \rightarrow C(n) = \frac{3}{2}n - 2 \end{array} \right.$$

每次三个（或四个）仍要比较这么多次，用分治法解决这个问题最终也需要约 $3n/2$ 次比较

查找第 k 小的数

问题：已知序列 $S = x_1, x_2, \dots, x_n$ 以及整数 k , $1 \leq k \leq n$, 试查找 S 中第 k 小的数。

如果 k 接近1或者 n ，那么只要把查找最小数的算法运行 k 次便能找到第 k 小的数。这个方法大概需作 kn 次比较。排序算法一般都比这个原始的算法好，除非 k 是 $O(\log n)$ 或者 $n-O(\log n)$ 。

解题思想大致与快速排序相同。在快速排序中，序列被主元分成两个子序列，然后分别对这两个子序列递归排序。而现在可先确定第 k 小的数在哪个子序列中，然后只要对那个子序列递归查找就行了。不必考虑其余的数。

```

Algorithm Selection(x,n,k)
Input: x (an array in the range 1 to n), and k (an integer)
Output: s (the kth smallest element, the array x is changed)
begin
    if (k<1) or (k>n) then print "error"
    else s:=Select(1,n,k)
end

Procedure Select(Left,Right,k);
begin
    if Left=Right then
        Select:=Left
    else
        Partition(x,Left,Right);
        Let Middle be the output of Partition;
        if Middle-Left+1>=k then Select(Left,Middle,k)
        else Select(Middle+1,Right,k-(Middle-Left+1))
end

```

复杂度： $O(n)$ ，最坏情况下第 k 小的为 min 或 max ，复杂度为 $O(n^2)$

数据压缩

问题：已知某个文本文件（字符串序列），为其设计一种编码方式使之满足前缀限制且使编码后的文本长度最短。

记字符为 C_1, C_2, \dots, C_n ，相应字频(每个字符在文本中出现的次数，通过对典型文本进行统计而得)为 f_1, f_2, \dots, f_n 。若某种编码方式E中，用长度为 s_i 的位串 S_i 表示 C_i 的话，文件F用编码方式E编码后的长度为 $L(E, F) = \sum s_i f_i$

目标：设计一种满足前缀限制的编码方式E，且 $L(E, F)$ 最小。

解码时按序扫描序列的每一位直到得到对应于某个字符编码的位串。为此考察二叉树，每个非叶节点有一条标着0、另一条标着1的两条发散边，每个叶子节点对应着一个字符，从根节点到叶子节点的路径就是这个字符的编码。当扫描文件的过程中到达叶子节点时，就能地确定相应的字符。

问题：如何构造这棵树，使 $L(E, F)$ 最小。

51

令 C_i 和 C_j 是字频最小的两个字符，则存在一棵使 $L(E, F)$ 最小的树，使 C_i 和 C_j 对应着离根节点最近的叶子节点。否则，若存在某个字频更高的字符在树上的位置低于这些字符，那么它就可以同 C_i 或 C_j 交换从而减小 $L(E, F)$ 。因为树上每个节点不是有两个子节点就是无子节点，于是可以断定 C_i 和 C_j 是两个并列的节点。

用一个新的字符代替 C_i 和 C_j ，这个新字符记作 C_{ij} ，其字频为 $f_i + f_j$ 。

现在问题有 $n-1$ 个字符($n-2$ 个原来的和一个新的)，这样就可以用归纳假设解决了。

```
算法 Huffman_Encoding(s, f)
输入: s (字符串) 和 f (字频数组)
输出: T (s的霍夫曼树)
根据字频将每个字符插入堆H;
while H不空 do
    if H只包含一个字符X then
        把X作为T的根节点
    else
        取字频最低的两个字符X和Y，从H中将它们删去;
        用新字符Z取代X和Y，且新字符Z的字频是X和Y的字频之和;
        把Z插入H;
        把X和Y作为T中Z的子节点{Z还没有父节点}
```

算法复杂度：构造树的过程中，每个节点都用常数时间。插入和删除操作分别用了 $O(\log n)$ 步。因此，算法总的运行时间是 $O(n \log n)$ 。

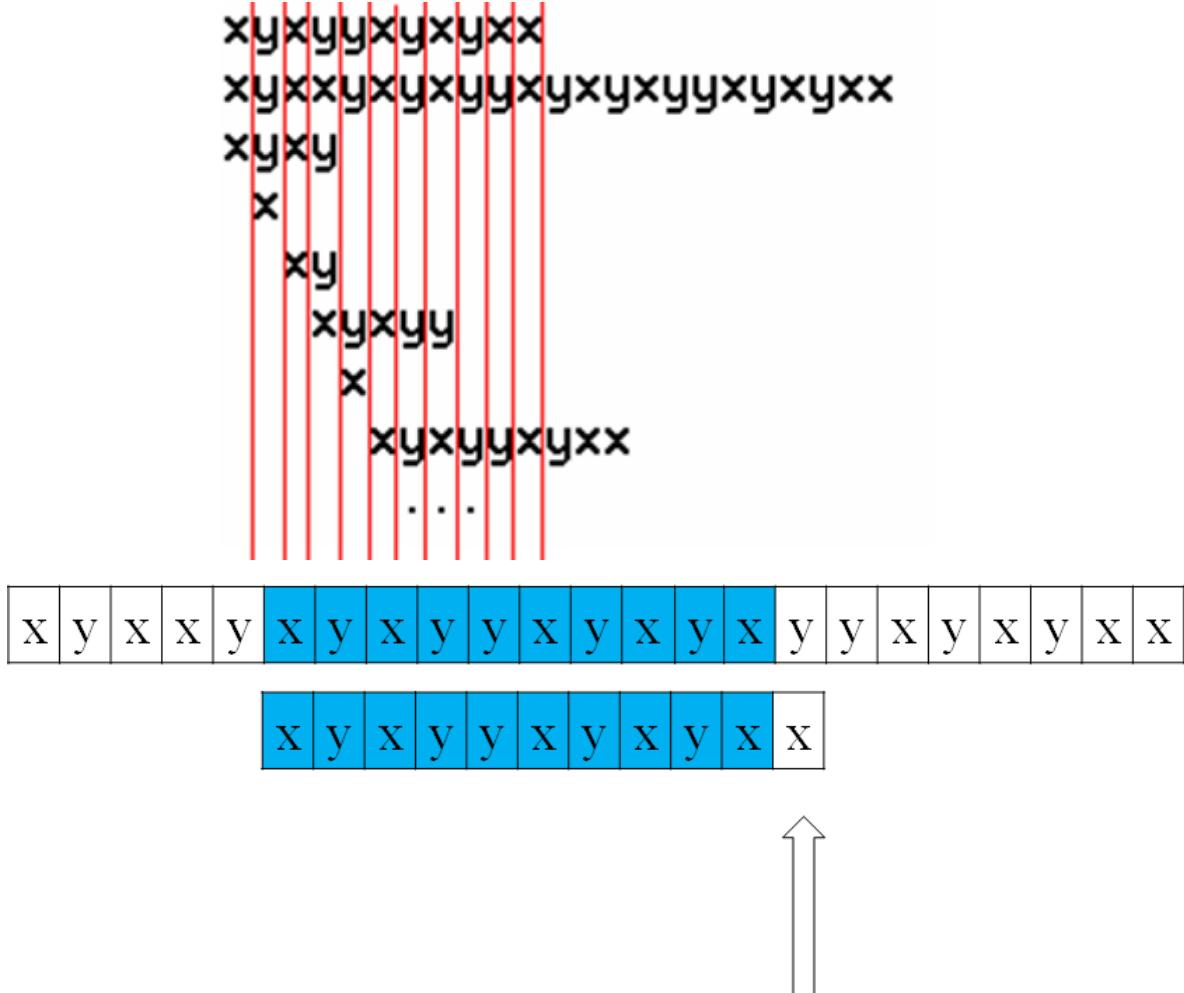
串匹配

问题：已知字符串A、B，在A中查找B的第一次出现(如果有的话)，即确定最小的k，使得对任意*i*， $1 \leq i \leq m$ ，有 $a_{k+i} = b_i$ 。

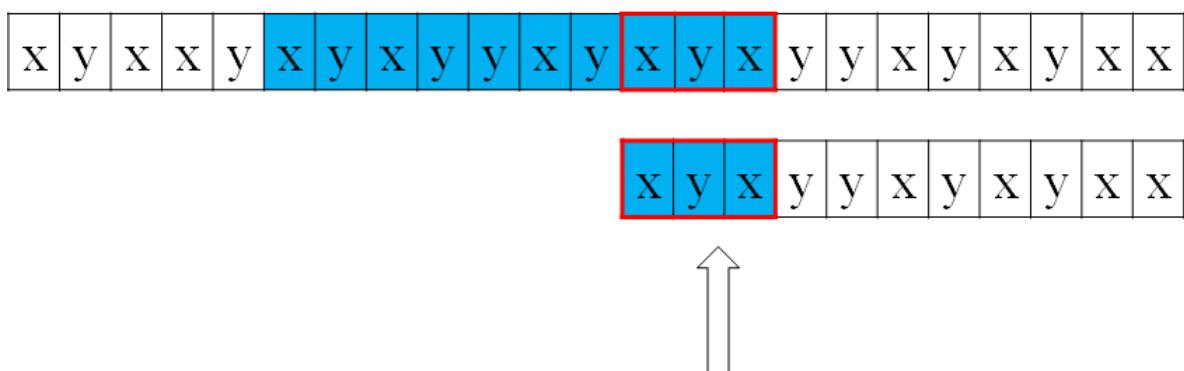
直接方法：从A的第一个能匹配b₁的字符开始，然后依次比较直到完全匹配或是中途遇到不匹配的字符。如果遇到了不匹配的字符，就必须开始重新进行匹配。

缺点：可能会回溯很多次，然后每次都重新开始匹配，导致最坏情况下的比较次数可能是O(mn)

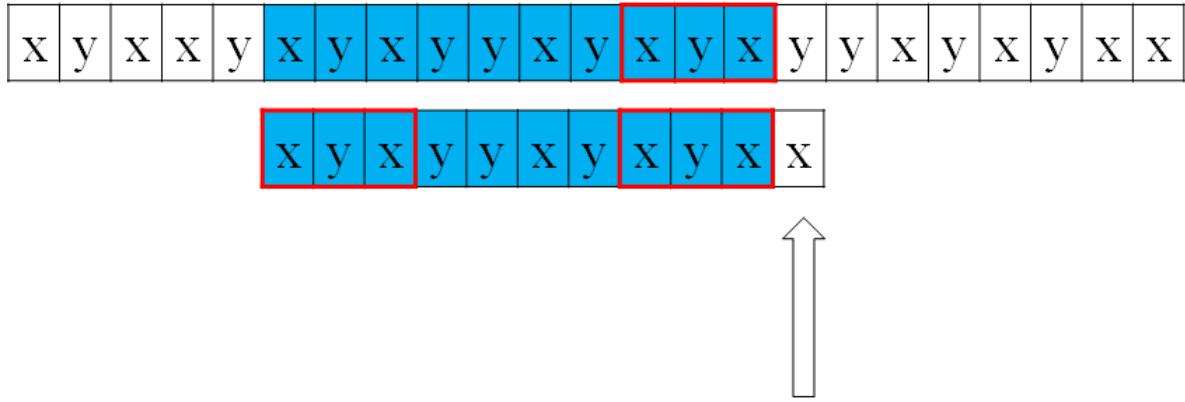
性能瓶颈：回溯。



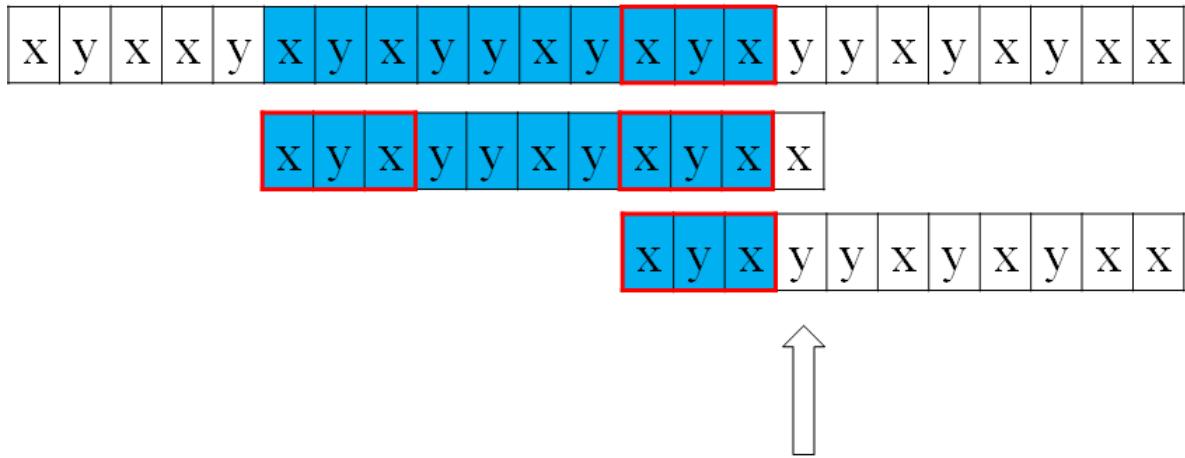
A的一段与B的开始一段匹配，
但B的第*i*位与A的对应位不匹配



B整体向右移动至B的前
面一段与A的一段匹配。



红色部分不需要再比较



下一次比较从B[4]开始
A不回溯

设计目标：即使发生不匹配也不必回溯的算法

Q: 已知 $B[1, \dots, q] = A[s+1, \dots, s+q]$, $B[q+1] \neq A[s+q+1]$

则满足以下条件的最小 s' 是多少?

$B[1, \dots, k] = A[s'+1, \dots, s'+k]$

$s'+k=s+q$

$s' > s$

A: $A[s'+1, \dots, s'+k]$ 是 $B[1, \dots, q]$ 的后缀

思想：对每个 b_i 都要计算出既是 $B[1, \dots, i-1]$ 的前缀又是后缀的字符串的最大长度 j ，则 A 中不匹配的那个字符就可以跳过多余的比较过程直接跟 b_{j+1} 比较，因为之前已经知道 A 中最近匹配的 j 个字符能跟 B 的前缀字符串匹配。

由于我们得到既是 $B[1, \dots, i-1]$ 的前缀又是后缀的字符串的最大长度，所以 B 再往左一位就不能跟 A 匹配了，不会错过匹配。

构造表 next : $\text{next}(i) = \text{使得 } B[i-j, \dots, i-1] = B[1, \dots, j] \text{ 的最大的 } j$ ($0 < j < i-1$)，如果这样的 j 不存在，则返回 0。定义 $\text{next}(1) = -1$, $\text{next}(2) = 0$

匹配过程如下：A的字符分别同B的字符一一比较，直到出现不匹配。假设不匹配发生在 b_i ，首先查next表，然后把A的当前字符同 $b_{\text{next}(i)+1}$ 比较（前 $\text{next}(i)$ 个字符已经匹配）。如果仍然不匹配，那么再同 $b_{\text{next}(\text{next}(i)+1)+1}$ 比较，以此类推直到匹配为止。

$x \ y \ x \ y \ y \ x \ y \ x \ y \ x \ x$ $-1 \ 0 \ 0 \ 1 \ 2 \ 0 \ 1 \ 2 \ 3 \ 4 \ 3$	$xyxxyyxyxyxx$ $xyxxxxyxyyxyxyxyxyxyxx$ $xyxy$ xy $xyxy$ $xyxxyyxyxyxx$ $xyxxyyxyxyxx$
---	--

```

Algorithm String_Match(A,n,B,m)  (KMP)
Input: A (a string of size n), B (a string of size m), and assume that next has
been computed
Output: Start (the first index such that B is a substring of A starting at
A[start])
begin
    j:=1; i:=1;
    Start:=0;
    while start=0 and i<=n do
        if B[j]=A[i] then
            j:=j+1; i:=i+1
        else
            j:=next[j]+1;
            if j=0 then
                j:=1; i:=i+1;
            if j=m+1 then Start:=i-m
    end

```

算法复杂度：A的某个字符可能要同B的数个字符比较。如果一次比较结果不匹配，则A的这个字符就和B的next表中指向的字符比较。如果结果仍然不匹配，就需要继续查表直到两者匹配或者遇到B的首字符。那么对A的某个字符，比如说 a_i ，究竟要回溯多少次呢？假设第一次不匹配发生在 b_k 处，由于每次回溯都往回移一位，因此最多回溯 k 次。而要扫描到 b_k ，实际上已经在无回溯的情况下向前移了 k 位。那么如果把回溯的开销算在前移上，则最多只需要把前移的开销加倍。这个问题中最多前移 n 次，故比较次数是 $O(n)$ 。

如何计算next表的值

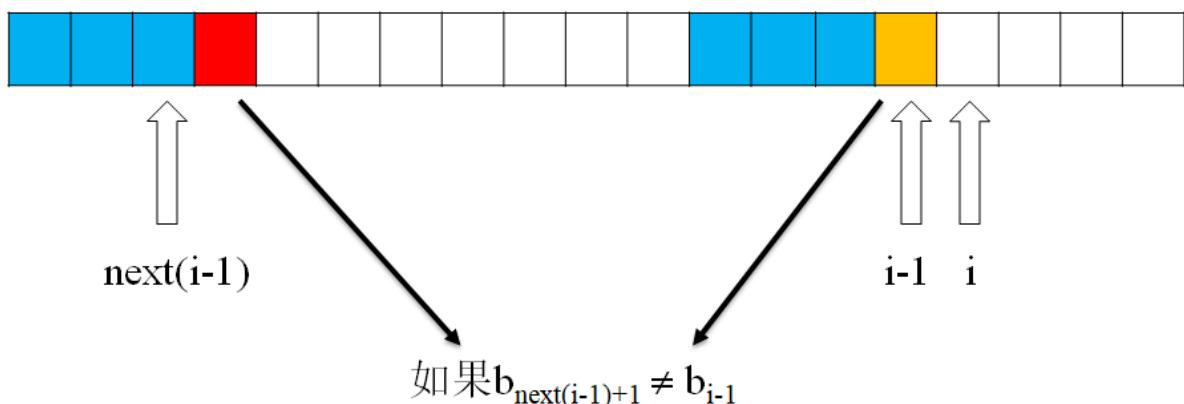
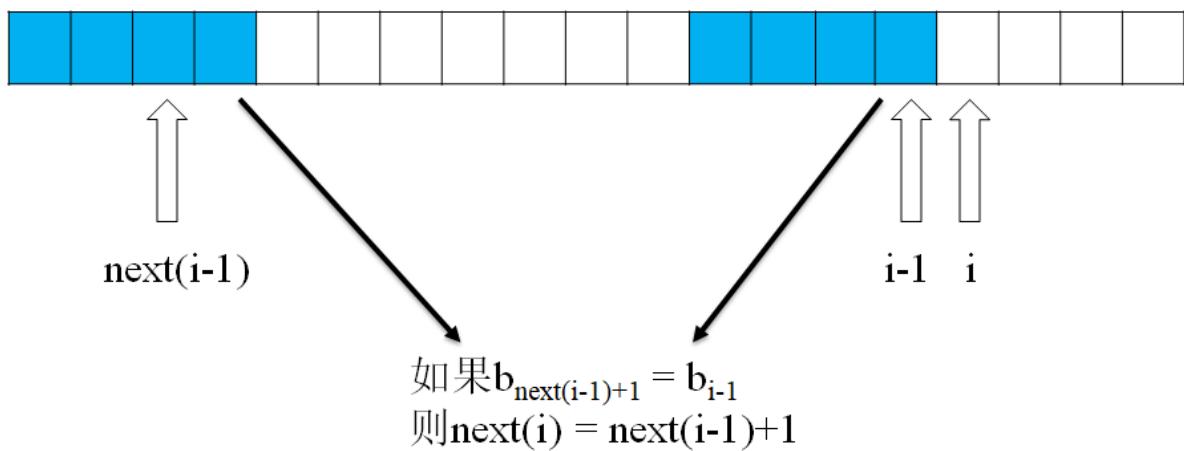
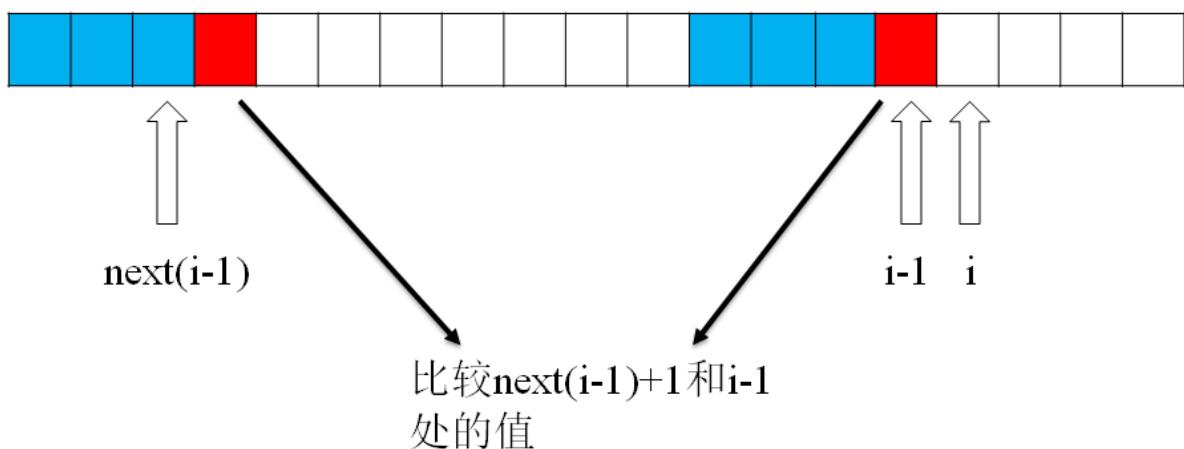
已知 $\text{next}(2) = 0$, 作为归纳基础。

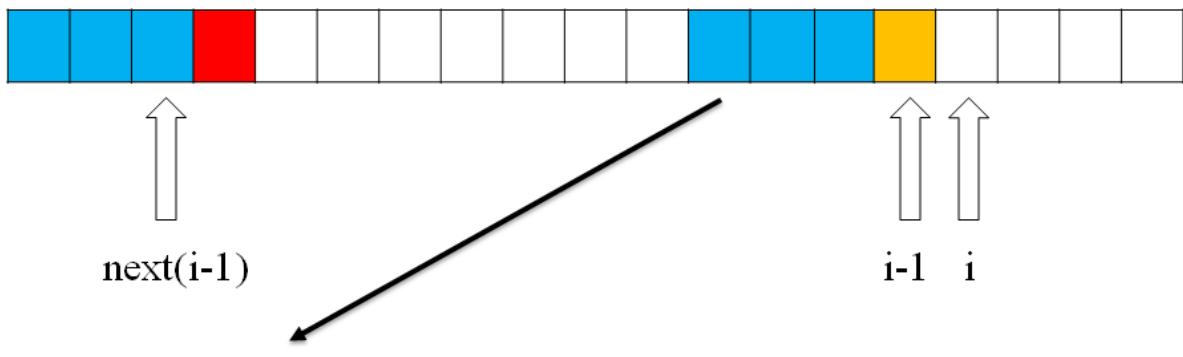
假设已经算出 $1, 2, \dots, i-1$ 的 next 值, 下面讨论如何计算 $\text{next}(i)$ 。

最佳情况下 $\text{next}(i)$ 等于 $\text{next}(i-1)+1$, 即 $b_{i-1} = b_{\text{next}(i-1)+1}$ 。

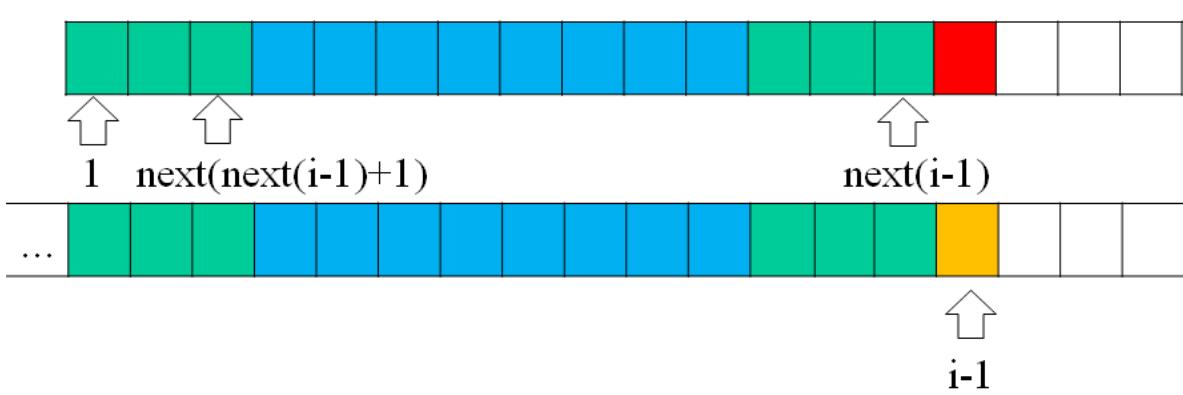
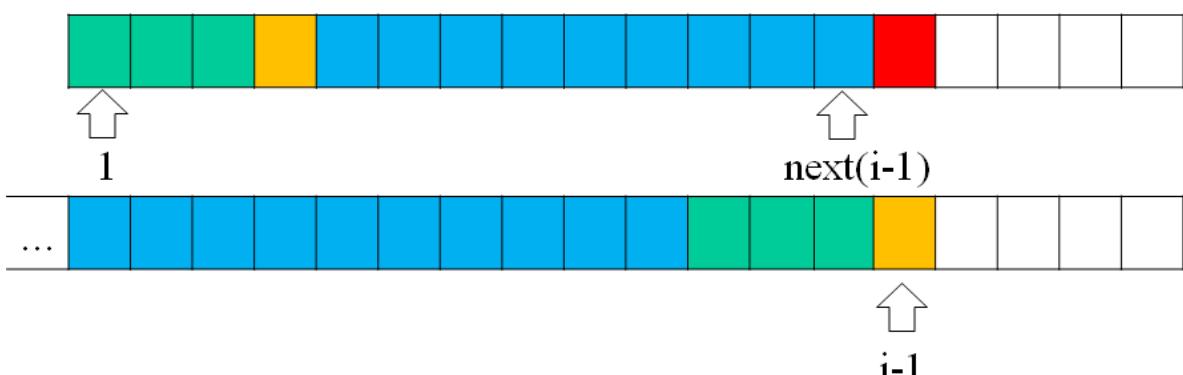
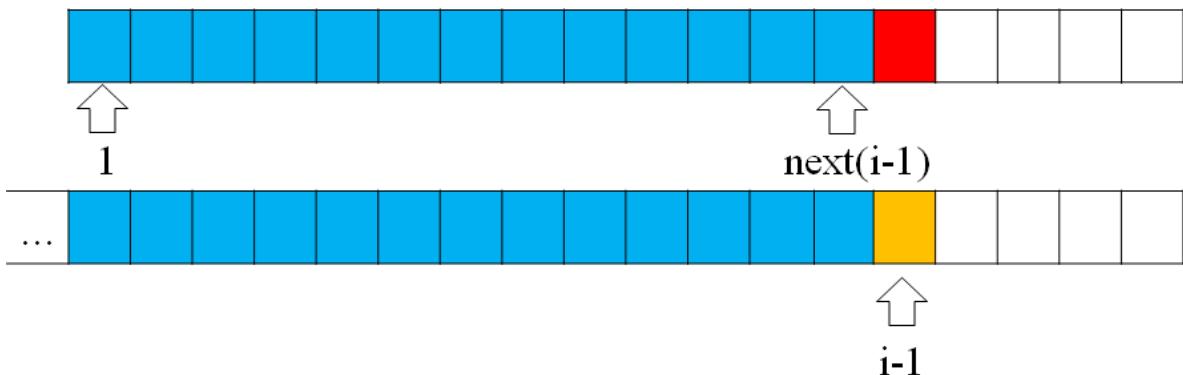
如果 $b_{i-1} \neq b_{\text{next}(i-1)+1}$, 首先要重新确定同前缀相等的新后缀。已知 $B[1, \dots, i-2]$ 的最大后缀同 $b[1, \dots, \text{next}(i-1)]$ 匹配, 而 $b_{i-1} \neq b_{\text{next}(i-1)+1}$ 恰恰意味着在 $b_{\text{next}(i-1)+1}$ 处发生了不匹配。因为如果在下标 j 处不匹配, 就去查 $\text{next}(j)$, 因此在 $\text{next}(i-1)+1$ 处不匹配自然就去查 $\text{next}(\text{next}(i-1)+1)$, 即试着匹配 b_{i-1} 和 $b_{\text{next}(\text{next}(i-1)+1)+1}$ 。如果它们能够相配, 那么 $\text{next}(i) = \text{next}(\text{next}(i-1)+1)+1$, 否则继续上述查找, 直到匹配或者回到起始处。

示例: 已知 $\text{next}(i-1)$, 求 $\text{next}(i)$

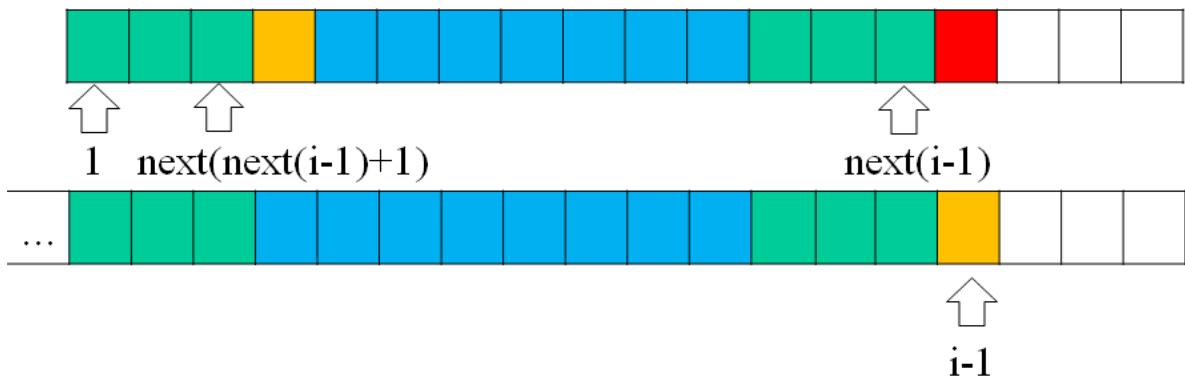




把这一部分移过来看

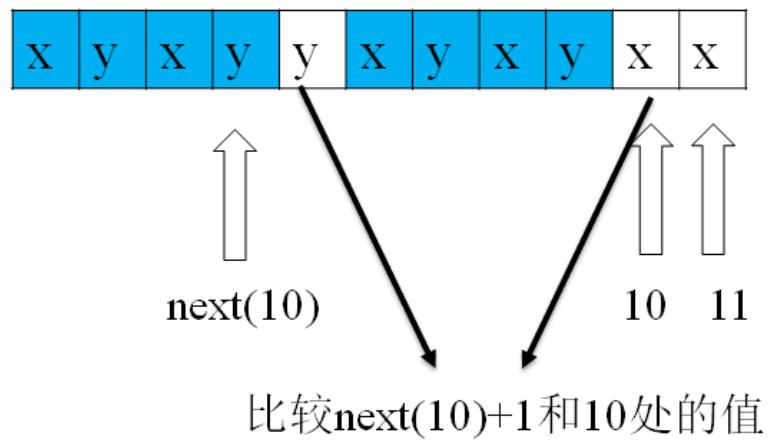


由上下相同及next的性质，四块绿色的相同

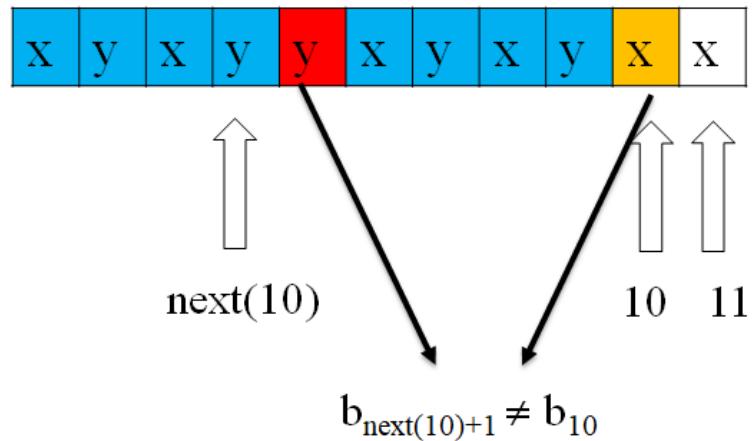


问题转化为比较 $b_{\text{next}(\text{next}(i-1)+1)+1}$ 和 b_{i-1} 是否相同
即两块黄色方格是否相同

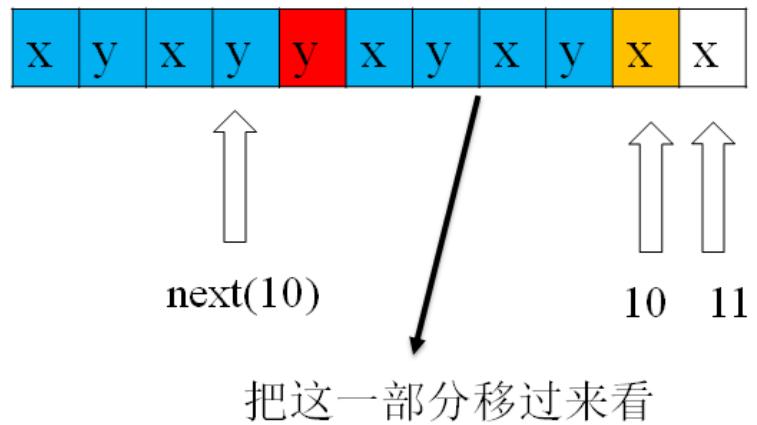
示例： $B=xyxxyy\ xyxyx\ x$, 已知 $\text{next}(10)=4$, 求 $\text{next}(11)$



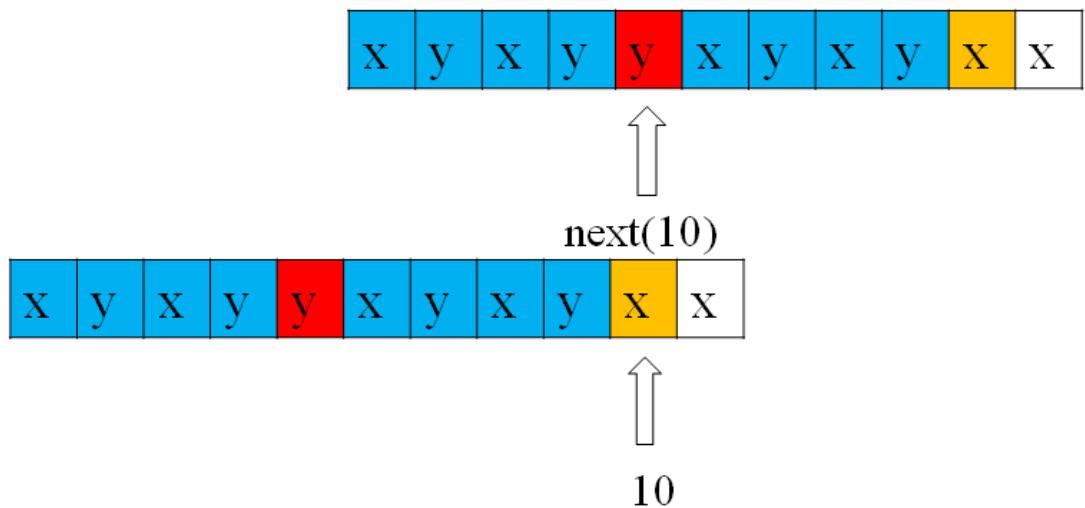
示例： $B=xyxxyy\ xyxyx\ x$, 已知 $\text{next}(10)=4$, 求 $\text{next}(11)$



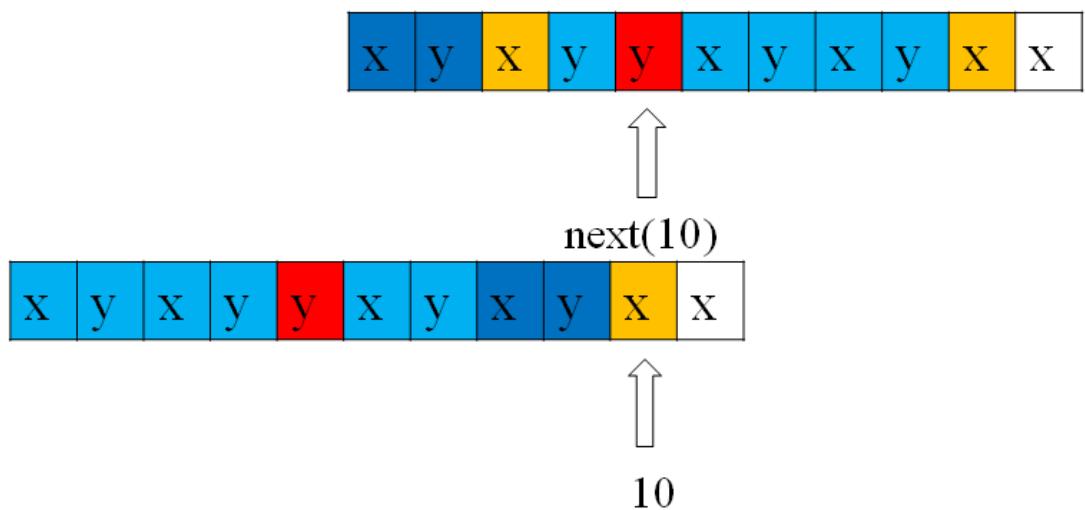
示例：B=xyxxyy xyxyyx x，已知next(10)=4，求next(11)



示例：B=xyxxyy xyxyyx x，已知next(10)=4，求next(11)

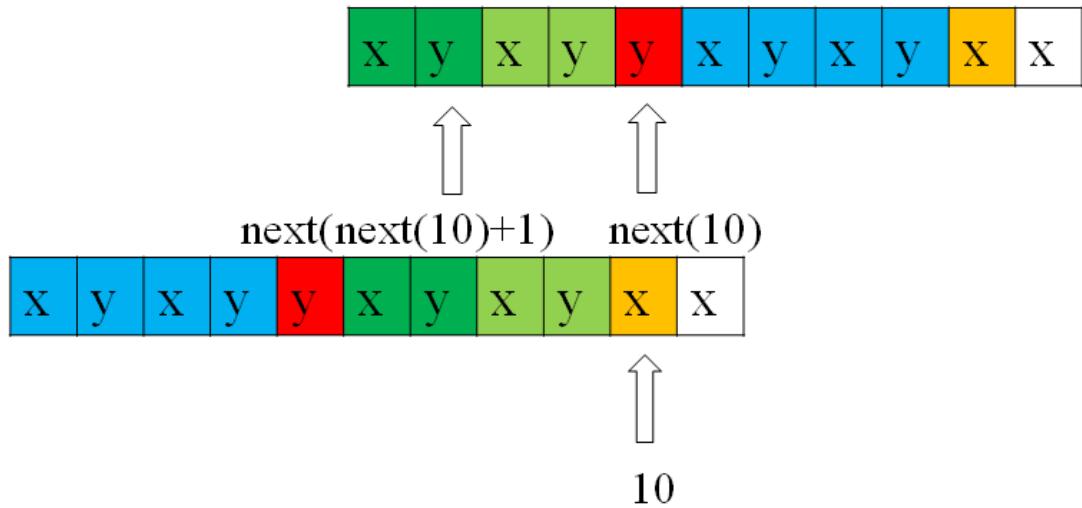


示例：B=xyxxyy xyxyyx x，已知next(10)=4，求next(11)



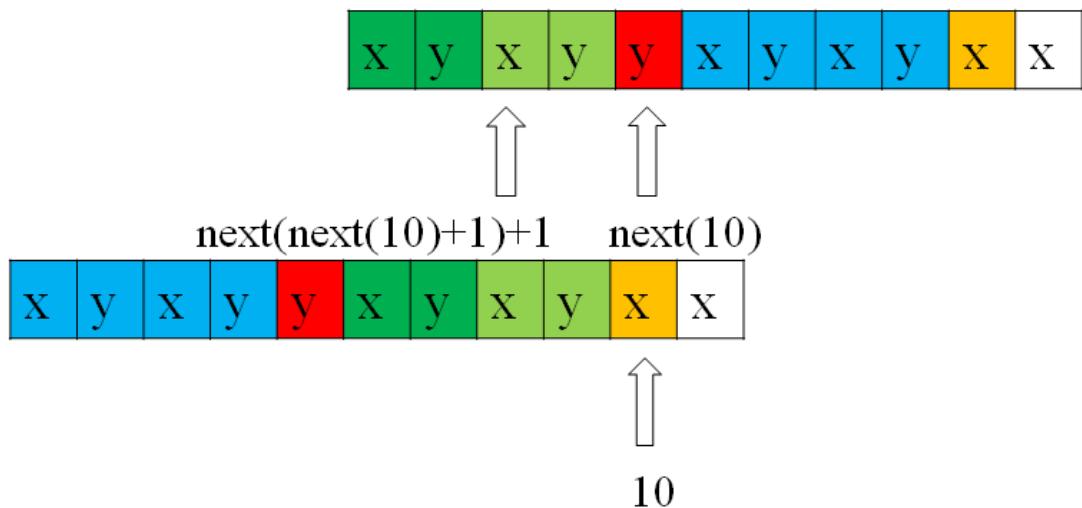
出现了如上情况

示例：B=xyxxyy xyxyyx x，已知next(10)=4，求next(11)



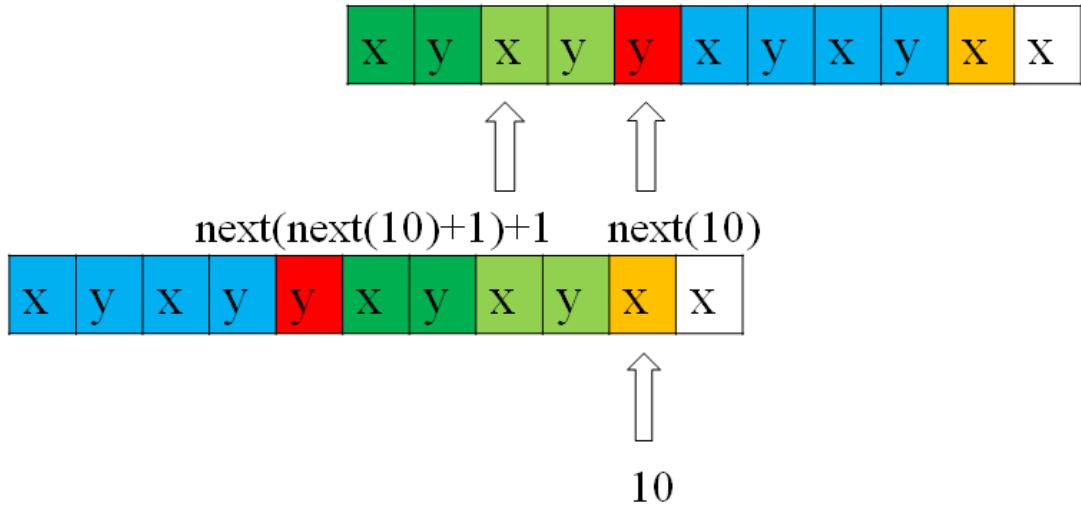
由上下相同及**next**的性质，四块绿色
(包括两块深绿两块浅绿) 的相同

示例：B=xyxxyy xyxyyx x，已知next(10)=4，求next(11)



问题转化为比较 $b_{\text{next}(\text{next}(10)+1)+1}$ 和 b_{10} 是否相同

示例：B=xyxxyy xyxyx x，已知next(10)=4，求next(11)



$b_{\text{next}(\text{next}(10)+1)+1}$ 和 b_{10} 相同，
因此 $\text{next}(11) = b_{\text{next}(\text{next}(10)+1)+1} + 1 = 3$

```
Algorithm Compute_Next(B,m)
Input: B (a string of size m)
Output: next (an array of size m)
begin
    next(1):=-1;
    next(2):=0;
    for i:=3 to m do
        j:=next(i-1)+1;
        while bi-1!=bj and j>0 do
            j:=next(j)+1;
        next(i):=j
end
```

算法复杂度：O(m)

方法：与前面KMP类似，关注j值的变化

序列比较

问题：把一个序列A=a1a2...an变成另一个序列B=b1b2...bm的最少修改步数

修改方法：1) 插入——把某个字符插入字符串，2) 删除——从字符串中删除某个字符，3) 替换——把某个字符替换成另一个字符。

记前缀子串 $a_1a_2\dots a_i(b_1b_2\dots b_i)$ 为 $A(i)(B(i))$ ，问题就转化为用最少的修改步数将 $A(n)$ 改为 $B(m)$ 。

记 $C(i, j)$ 为 $A(i)$ 变成 $B(j)$ 的最小代价，建立 $C(n, m)$ 和某个 $C(i, j)$ 之间的关系，这里 i, j 小于 n, m 。

删除：如果 A 到 B 的最小改动需要删除 a_n ，那么最佳方法就是先把 $A(n-1)$ 变成 $B(m)$ ，然后再删除这个字符。换句话说， $C(n, m) = C(n-1, m) + 1$ 。

插入：如果 A 到 B 的最小改动需要插入某个字符和 b_m 匹配，那么就有 $C(n, m) = C(n, m-1) + 1$ 。也就是（通过归纳）先把 $A(n)$ 以最小代价变成 $B(m-1)$ ，然后再插入一个跟 b_m 相同的字符。

替换：如果用 a_n 替换 b_m ，那么首先要找到从 $A(n-1)$ 到 $B(m-1)$ 的最小改动，接下来若 $a_n \neq b_m$ ，还要加1。

匹配：如果 a_n 和 b_m 相同，则 $C(n, m) = C(n-1, m-1)$ 。

$$c(i, j) = \begin{cases} 0 & a_i = b_j \\ 1 & a_i \neq b_j \end{cases}$$

$$C(n, m) = \min \begin{cases} C(n-1, m) + 1 & \text{delete } a_n \\ C(n, m-1) + 1 & \text{insert for } b_m \\ C(n-1, m-1) + c(n, m) & \text{replacing V matching } a_n \end{cases}$$

$$C(i, 0) = i$$

$$C(0, j) = j$$

把一个规模为 (n, m) 的问题归纳到三个规模仅比原问题略小的子问题。

总的子问题数并不多。不会超过 nm 。

使用动态规划

```

Algorithm Minimum_Edit_Distance(A, n, B, m)
Input: A (a string of size n), and B (a string of size m)
Output: C (the cost matrix)
Begin
    for i:=0 to n do C[i, 0]:=i;
    for j:=1 to m do C[0, j]:=j;
    for i:=1 to n do
        for j:=1 to m do
            x:=C[i-1, j]+1;
            y:=C[i, j-1]+1;
            if ai=bj then z:=C[i-1, j-1]
            else z:=C[i-1, j-1]+1;
            C[i, j]:=min(x, y, z);
    end

```

时间复杂度为 $O(nm)$ ，空间复杂度为 $O(nm)$

只依赖当前行和上一行，所以可以只保存两行，把 i 换 $i \% 2$ ， $(i-1)$ 换 $(i-1) \% 2$ ，空间复杂度为 $O(m)$

查找众数

令E是整数序列x₁,x₂,...,x_n。E中x的重数是x在E中出现的次数。如果某个数z的重数大于n/2，则它就是E中的众数 (majority)。

问题：已知一数列，判断是否存在众数。若存在，则求出众数。

蛮力搜索 $O(n^2)$ ，对每个数比较其他数是否与之相同

排序算法：排序后进行统计。 $O(n \log n)$ ，排序用时 $O(n \log n)$ ，排序后查找用时 $O(n)$

中数查找法：如果众数存在，它肯定等于中数（中数是第(n/2)小的数，而众数的出现次数超过n/2）。所以一旦找到中数，就能统计它出现的次数。

删除两个不同的数后，原来集合中的众数仍为新集合中的众数。（否命题不成立：1、2、5、5、3中没有众数，但如果删除1和2，5就成了新的众数。）

按出现的顺序依次扫描。引入变量C（候选者）和M（重数）。当考察x_i时，C是x₁,x₂,...,x_{i-1}中众数的候选者，M等于C当前已出现的次数减去被删除的次数，即x₁,x₂,...,x_{i-1}被分成大小分别为2k和M的两组，其中2k+M=i-1。第一组有k对不同的数，第二组包含已至少出现M次的C。如果在x₁,x₂,...,x_{i-1}中存在众数，在不断删除后，它必定能胜出并等于C。（C不是众数，也有可能在删除后胜出。）考察x_i，将它同C比较，根据比较结果增加或减少C出现次数。若不存在候选的众数（M=0），则C=x_i，M=1。最后只剩一个候选者C，统计其出现次数便能判定它是否为众数。

```
Algorithm Majority(X,n)
Input: X (an array of size n of positive numbers)
Output: Majority (the majority in X if it exists, or -1 otherwise)
begin
    C:=X[1];
    M:=1;
    for i:=2 to n do
        if M=0 then
            C:=X[i];
            M:=1;
        else
            if C=X[i] then M:=M+1
            else M:=M-1;
        if M=0 then Majority:=-1
        else
            Count:=0;
            for i:=1 to n do
                if X[i]=C then Count:=Count+1;
            if Count>n/2 then Majority:=C else Majority:=-1;
    end
```

算法复杂度：找到候选者需n-1次比较，最坏情况下判定该候选者是否为众数也需要n-1次比较。因此，总共需2n-2次比较。

最长递增序列

给定一个由不同整数组成的序列，求其最长递增序列LIS

归纳假设1：给定某个长度小于m的序列，知道如何求它的某个最长的递增序列

归纳假设2：给定某个长度小于m的序列，知道如何求它的所有最长的递增序列

产生新问题：

若 x_m 不能使所有 $LIS(m-1)$
变长，但考虑到
新的 LIS (同长度的)

78945612 3

还要求出第二长的递增序列，同样这样也可能需要求出第三长的递增序列，因为其可能变成第二长的

归纳假设3：给定某个长度小于 m 的序列，知道如何求它的某个最长的递增序列，使得其它最长递增序列的末尾的数都不比这个序列末尾的数小

原因：因为这样变长的可能性最大，如 567123 中选择 123

问题：可能存在新元素比末尾数小

$BIS.last > x_m$

$L + x_m \rightarrow$ 新的 BIS

例如 567124 中添加 3，这样 123 可以替代 124

面临和 1 同样的问题：

长度为 $s-1$ 的 $LIS + x_m \rightarrow BIS$

归纳假设4：给定某个长度小于 m 的序列，知道如何对任意 $k < m-1$ 求出 $BIS(k)$ ，如果存在的话

要得到更好的 $BIS(k+1)$

x_m 能力加长某个 $BIS(k)$

$\left\{ \begin{array}{l} \text{长度为 } k \text{ 的最长 } LIS \\ \Leftrightarrow \begin{array}{l} 1) x_m > BIS(k).last \text{ 可加入} \\ 2) x_m < BIS(k+1).last \text{ 可能性} \uparrow \text{更好} \end{array} \end{array} \right.$

$BIS(1).last < BIS(2).last$
 $< \dots < BIS(k).last$

```
给定xm，设前m-1个数字的LIS长度为s
for i=s downto 1
    if BIS(i).last < xm      break
end for
if i=0  then BIS(1)=xm
if i=s  then BIS(s+1)=BIS(s)+xm
if 0 < i < s then BIS(i+1)=BIS(i)+xm  (即BIS(i).last < xm < BIS(i+1).last)
```

其中i的查找可用二分查找

所以每个xm需要O(logm)次比较

总的运行时间为O(nlogn)

查找集合中最大的两个元素

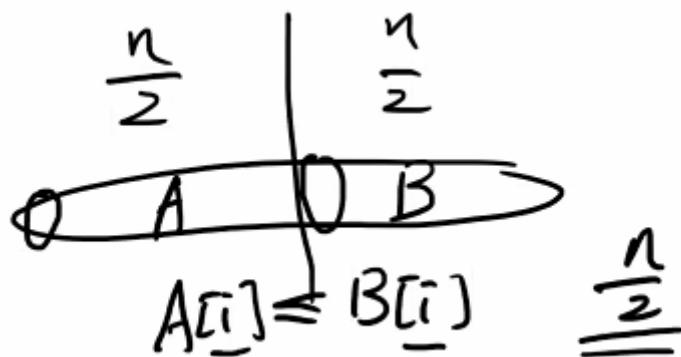
给定有n个元素的集合S，求其中最大的和第二大的元素

直接法：2n-3

常规的分治法：3n/2-2

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

多花一些分的时间的分治法：n+logn-2

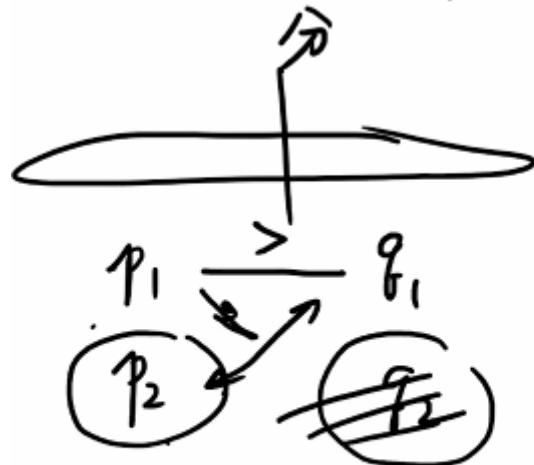


- B中找出最大和第二大的元素，则B (max1) 为n个元素中最大的，次大元素时B (max2) 和 A (max1) 中较大者

$$T(n) = \frac{n}{2} + T\left(\frac{n}{2}\right) + 1$$

分 分 合

第二大元素的候选者集合：n+logn-2



$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

p_1 和 q_1 比较，假设 p_1 赢了，淘汰 q_2 ，将 q_1 和 p_2 加入候选者集合

候选者集合规模为 $\log n$

总共比较 $n + \log n - 1$

计算多重集合的模

求给定的多重集合(元素不必互异)S的模，即出现最多的元素

排序 $O(n \log n)$

直接归纳

分治法(利用中数): $O(n \log n)$

- 按中数划分为三部分，其中大于和小于中数的两个集合中元素个数不会超过 $n/2$

$$\cdot T(n) \leq 2T\left(\frac{n}{2}\right) + \underline{O(n)}$$

分治法(改变归纳基础): $O(n \log(n/M))$

- 更改递归结束条件
- 设模的重数为 M

$$T(M) = O(M)$$

$$\cdot \underline{T(n) = O(n \cdot \log(n/M))}$$

频繁元素-- Misra-Gries 算法

频繁元素是指输入数据中有些元素会出现多次，希望找到出现最频繁的 t 个

如果输入数据不是很多，可以对每一个不同的元素设立一个计数器，数据输入时，相应的计数器加1。这样的算法是一个精确的算法，但需要的内存量可能会比较大，尤其是数据种类很多时，并且输入数据的数量也很多时，这一矛盾就会更加突出。

Misra-Gries算法是一种计算频繁元素的近似算法，该算法需要事先规定计数器的最大数量k，即只在内存中保留k个计数器。当有数据输入时，检查是否已经为其分配了计数器，这时共有三种不同的情况需要处理：

- 1) 有对应的计数器，则相应的计数器加1
- 2) 没有相应的计数器，并且已有的计数器数量小于k，则为该数据分配一个新的计数器，其值设为1。
- 3) 没有相应的计数器，并且已经有了k个计数器，则将所有的计数器数值减1，并删除值为0的计数器，新数据也被丢弃。

例子

输入：1, 5, 3, 1, 2, 1, 2, 3, 7, 1

k=3，则计算过程为：

输入	计数器<数据, 次数>
1	<1,1>
5	<1,1> <5,1>
3	<1,1> <5,1> <3,1>
1	<1,2> <5,1> <3,1>
2	<1,1>
1	<1,2>
2	<1,2> <2,1>
3	<1,2> <2,1> <3,1>
7	<1,1>
1	<1,2>

性能

Misra-Gries算法是一种可以用于寻找频率大于 $n/(k+1)$ 的元素的空间亚线性的近似算法。

如需获得准确的结果，则需要把这些算法结束时还留在内存中的计数器置0，再次扫描所有数据，并对相应计数器做计数工作。第二轮计数完毕后，检查各个计数器的值是否超过 $n/(k+1)$ 。但两次扫描也意味着需要存储所有的数据，不再是一个空间亚线性的算法了。

频繁问题的一个特例是找出出现次数超过一半的元素，即寻找众数，只需将Misra-Gries算法中的k设为1即可

假设可能出现的数字m种
全 $\{1, 2, \dots, m\}$

输入 n 个 数为 a_1, a_2, \dots, a_n
 $f(i)$ 表示其中已出现的次数

$$\sum_{i=1}^m f(i) = n$$

$f'(i)$ 算法结束时 i 对应计数器的次数
不在内存中的数据认为计数器次数为 0

显然 $f'(i) \leq f(i)$

$$\sum_{i=1}^m f'(i) = n' \leq n$$

在第 3 种情况下，计数器值出现减 1，相当于所有元素的出现次数减少了 $(k+1)$

\therefore 第 3 种情况下出现次数 $\frac{n - n'}{k+1}$

$$\therefore f(a_i) - f'(a_i) = \frac{n - n'}{k+1}$$

$$f'(a_i) \leq f(a_i) \leq f'(a_i) + \frac{n - n'}{k+1}$$

$$\begin{aligned} f'(a_i) &\geq f(a_i) - \frac{n - n'}{k+1} \\ &\geq f(a_i) - \frac{n}{k+1} \end{aligned}$$

若某个 a_i 出现的次数大于 $\frac{1}{k+1}$
 $f'(a_i) > 0$

Ch 07 图算法

基本概念

本章在大部分情况下将使用邻接表来表示图，这对稀疏图(即，图中的边相对较少)比较有效。

一个图 $G=(V, E)$ 包括一个顶点(也称之为节点)的集合 V 和一个边的集合 E 。每一条边对应一对不相同的顶点。(有时候自循环也是允许的，即一个顶点到自身有一条边；但现在假定不允许这种情况。)

一个图可以是有向的或者无向的。有向图中的边是一个有序对；边所连接的两个顶点的次序是重要的。无向图中的边是一个无序对。

若相同的顶点对之间有若干条边，则对应的图称为多重图(即 E 为多重集)。不是多重图的图有时称之为简单图。除非特别指明，本章假定所处理的图都是简单的。

一个顶点 v 的度 $d(v)$ 是与 v 相连接的边的数目。对于有向图，要区分入度和出度，前者是以 v 为头部的边的数目，后者是以 v 为尾部的边的数目。

从 v_1 到 v_k 的一条路径是顶点序列 v_1, v_2, \dots, v_k ，它们通过边 $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ (这些边通常被看作是路径的一部分)相连接。如果每一个顶点在路径中至多出现一次，则这条路径被称之为简单的。如果从顶点 v 到顶点 u 存在一条路径，则称 u 是可达的。回路是一条起点与终点为同一顶点的路径。若除了第一个和最后一个顶点，其它顶点至多只出现一次，则此回路被称之为简单的。一个简单的回路也称之为闭链。(有时候即使回路不是简单的也被称之为闭链；但本章假定闭链是简单的回路。)

一个有向图 $G=(V, E)$ 的无向型是一个与 G 相同的图，但其中的边没有方向性。对于一个图的无向型，若任意两个顶点之间都存在一条路径，则图被称之为连通的。一个森林是一个不包含闭链的(无向型)图。

一棵树则是一个连通的森林。一棵根树(也叫树形图)是一个有向树，其中一个特定顶点被称之为根，所有的边都远离这个根延伸。

图 $G=(V, E)$ 的一个子图是图 $H=(U, F)$ ，其中 $U \subseteq V, F \subseteq E$ 。无向图 G 的一个生成树是 G 的一个子图，它是一棵树并且包含 G 中所有顶点。无向图 G 的一个生成森林是 G 的一个子图，它是一个森林并且包含 G 中所有顶点。图 $G=(V, E)$ 的一个顶点导出子图(简称为导出子图)是一个子图 $H=(U, F)$ ，其中 $U \subseteq V$ 且 F 包含所有两个顶点在 U 中的 E 中的边。

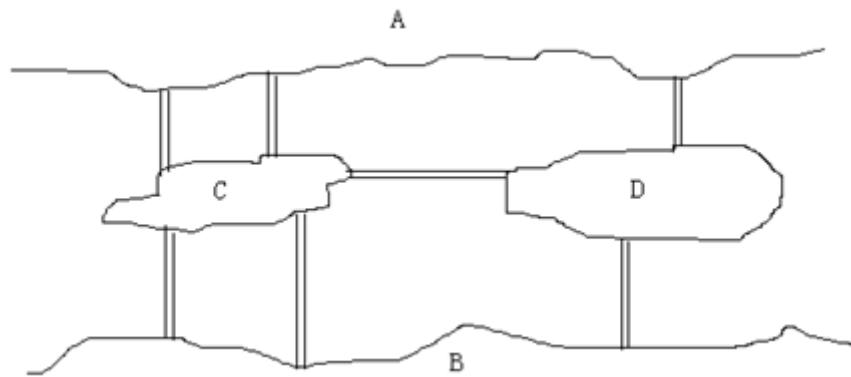
如果图 $G=(V, E)$ 不是连通的，则可以用唯一的方式把它划分成一些连通子图的集合，这些连通子图被称之为 G 的连通分支。图 G 的一个连通分支是一个连通子图，且不属于任何其他连通子图，即连通分支是一个最大连通子图。

一个偶图是一个顶点集合可以划分成两个集合的图，且图中所有边的两个顶点都分属不同的点集，即同一集合的顶点之间没有边。

一个加权图是边附有权重的图。

欧拉图

哥尼斯堡七桥问题



问题：给定一个无向连通图 $G=(V, E)$ ，其所有顶点的度数为偶数，能否找到一条封闭路径 P ，使得 E 中的每一条边在 P 中仅出现一次。

从数学归纳法角度

存在这种封闭路径 \Rightarrow 图连通
所有顶点度数
为偶数

\Rightarrow 也存在封闭路径
遍历时，每个顶点进出的
次数相同， \because 每条边经过一次
 \therefore 与每个顶点相连的边数为偶数

\Leftarrow 设当图的边数 $< m$ 时命题成立
当 $|E|=m$ 时 G 中存在回路 P
(不包含所有边)

$G - P \rightarrow G'$ 上的度数
还是偶数

G' 可能由多个连通分支组成
 每个连通分支 G'_i 存在所要求的
 回路 P_i
 由这些 P_i 和 P 组合构成欧拉回路

图的遍历：深度优先搜索(DFS)

对于一个用邻接表表示的无向图，遍历是从任意一个顶点 r 开始，它称之为DFS的根，这个根被标记为被访问过。然后，从与 r 连接的顶点中任意挑一个(未被标记过的)顶点 r_1 ，再从 r_1 开始(递归)执行DFS。当到达一个顶点 v ，它的所有连接顶点都已经被标记过了，则递归停止。如果当对 r_1 进行DFS终止后，所有与 r 连接的顶点都已经被标记过了，则对 r 进行的DFS终止。否则，在与 r 连接的顶点中任选一个未被标记过的顶点 r_2 ，从 r_2 开始进行DFS，等等。

```

Algorithm Depth_First_Search(G,v);
Input: G=(V,E) (an undirected connected graph), and v (a vertex of G)
Output: depends on the application
begin
    mark v;
    perform preWORK on v; {preWORK depends on the application of DFS}
    for all edges (v,w) do
        if w is unmarked then Depth_First_Search(G,w);
        perform postWORK for (v,w);
        {postWORK depends on the application of DFS; it is sometimes performed
        only on edges leading to newly marked vertices}
    
```

引理 7.1：如果 G 是连通的，则通过深度优先搜索可以对它的所有顶点进行标记，并且在算法的执行过程中，它的每条边至少被查看过一次。

连通分支划分

```

Algorithm Connected_Components(G)
Input: G=(V,E) (an undirected graph)
Output: v.Component is set to the the number of the component containing v, for
every vertex v
begin
    Component_Number:=1;
    while there is an unmarked vertex v do
        Depth_First_Search(G,v);
        (using the following preWORK: v.Component:=Component_Number;)
        Component_Number=Component_Number+1
end
    
```

复杂性：每条边恰好被查看过两次(从每一端点各一次)，所以，运行时间与边的数目成比例。由于图中可能包含一些与其它点不相连的顶点，所以在运行时间的表达式中必须包括 $O(|V|)$ ，因此总共的运行时间是 $O(|V| + |E|)$ 。

按DFS访问顺序做编号

```

Algorithm DFS_Numbering(G, v);
Input: G=(V, E) (an undirected graph), and v ( a vertex of G)
Output: for every vertex v, v.DFS is set to the DFS number of v
Initially DFS_Number:=1;
USE DFS with the following preWORK:
preWORK:
    v.DFS:=DFS_Number;
    DFS_Number:=DFS_Number+1;

```

构造DFS树

```

Algorithm Build_DFS_Tree(G, v);
Input: G=(V, E) (an undirected graph), and v ( a vertex of G)
Output: T (a DFS tree of G; T is initially empty)
USE DFS with the following postWORK:
postWORK:
    if w was unmarked then add the edge (v, w) to T;
    {the statement above can be included in the if statement of algorithm
Depth_First_Search}

```

无向DFS树的主要性质

在一个以r为根的树中，如果顶点v位于从顶点w到r的一条唯一路径上，则v被称之为w的祖先。若v是w的一个祖先，则w被称之为v的后代。

引理 7.2：令G=(V, E)是一个连通的无向图，T=(V, F)是由Build_DFS_Tree算法构造出的G的一个DFS树，则E中的每一条边e，或者属于F(即 $e \in F$)，或者连接G中的两个顶点，它们其中一个是T中的另一个的祖先。

$\forall (u, v) \in E$
在DFS中 u 早于 v 被访问
当 u 被标记后，会对所有
 u 的未访问邻居进行DFS
 $\therefore v$ 是 u 的邻居
 \therefore DFS 可以从 v 开始 $\Rightarrow (u, v) \in F$
也可以在回溯至 u 之间先访问了
 v ，此时 v 为 u 的后代

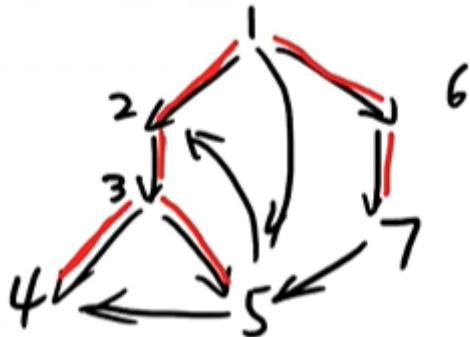
有向图的DFS

有向图的DFS过程与无向图一样。

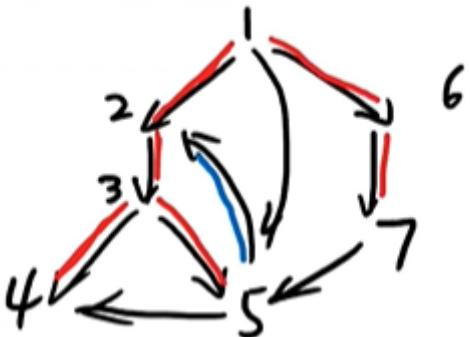
有向图进行DFS后有四种类型的边——树边，后退边，前向边，和交叉边。前三类边连接两个顶点，其中一个是另一个在树中的后代：树边把树中的parents连接到children，后退边把后代连接到祖先，而前向边把祖先连接到后代。只有交叉边连接在树中无“关联”的两个顶点，且必须“从右到左”交叉。

引理 7.3：令 $G=(V,E)$ 是一个有向图，且令 $T=(V,F)$ 是 G 的DFS树。如果 (v,w) 是 E 中的一条边，满足 $v.\text{DFS_Number} < w.\text{DFS_Number}$ ，则 w 是 v 在树 T 中的一个后代。

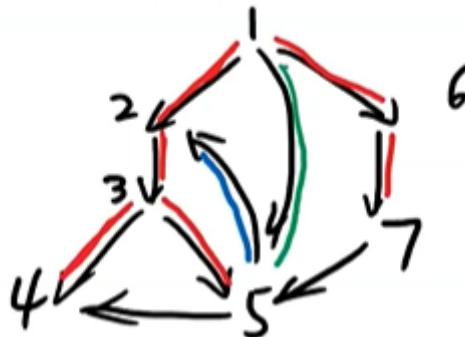
树边



后退边



前向边



交叉边

剩余的边，两个点没有祖先关系，都是从右到左的

无向图中有树边和连接祖先和子孙的无向边

对于连通无向图的DFS，从任意顶点开始，即可遍历整个图。但对于有向图则非如此，须再次从一个未被标记的顶点开始，一直继续下去直到所有的顶点被标记。



有向图闭链问题

问题：给定一个有向图 $G=(V, E)$ ，确定它是否含有一个(有向)闭链。

引理7.4：令 $G=(V, E)$ 是一个有向图，且令 T 是 G 的一个DFS树。则当且仅当 G 含有一个(相对于 T 的)后退边时， G 含有一个有向闭链。

若存在回路 C , ~~很想~~
 设 v 是 C 上 DFS 数最小的上
 指向 v 的边 (w, v) 必为后退边
 $w.DFS > v.DFS$
 \therefore 不可能 \exists 前向边或树边

若 v 不是 w 在树中祖先
 设 u 是 v 和 w 的最低的共同祖先
 $\because v.DFS < w.DFS$
 $\therefore v$ 在 v 的子树中且在包含 w
 的 u 所在的子树前被访问

* v 在 u 的子树中

这意味着 v 到 w 的唯一路径
 是经过 u 或 u 的一个祖先
 但 C 含一条从 v 到 w 的路
 $\therefore v$ 是 C 中 DFS 最小的 $\therefore C$ 中不能
 有 w 的祖先

```

Algorithm Find_a_Cycle(G);
Input: G=(V,E) (a directed graph)
Output: Find_a_Cycle (true if G contains a cycle and false otherwise)
Use DFS, starting from an arbitrary vertex, with the following preWORK and
postWORK:

preWORK:
    v.on_the_path:=true;
    {x.on_the_path is true if x is on the path from the root to the current
    vertex}
    {initially x.on_the_path=false for all vertices, and Find_a_Cycle is false}
postWORK:
    if w.on_the_path then Find_a_Cycle:=true; halt;
    if w is the last vertex on v's list then v.on_the_path:=false

```

图的遍历：广度优先搜索(BFS)

从顶点v开始，首先访问v全部的儿子，接着访问全部的“孙子”，依次类推。如果顶点w是通过BFS被标记的第k个顶点，则其具有的BFS数为k。通过只保留指向新访问顶点的边，可以构造一个BFS树。

```

Algorithm Breadth_First_Search(G,v);
Input: G=(V,E) (an undirected connected graph), and v (a vertex of G)
Output: depends on the application
begin
    mark v;
    put v in a queue {First In First Out};
    while the queue is not empty do
        remove the first vertex w from the queue;
        perform preWORK on w; {depends on the application}
        for all edges (w,x) such that x is unmarked do
            mark x, add (w,x) to the tree T, put x in the queue
end

```

性质

引理 7.5：如果边(u, w)属于一个BFS树，其中u是w的父母，则在具有导向w的边的顶点中，u具有最小的BFS数。

引理 7.6：对于每一个顶点w，在T中从根到w的路径是在G中从根到w的最短路径。

顶点w的层数是在树中从根到w的路径长度。BFS是逐层对图进行遍历。

引理 7.7：如果(v, w)是E中的一条边且不属于T，则它连接的两个顶点的层数至多相差1。

拓扑排序

问题：给定一个有向非循环图G=(V, E)，它有n个顶点，把顶点按照1到n做标记，满足，若v被标记为k，则通过有向路径从v出发可以到达的所有顶点的标记>k。

只有一个顶点的基础情形是平凡的。

归纳假设：已知如何根据条件对顶点数<n的有向非循环图做标记。

引理 7.8：一个有向非循环图总有一个入度为0的顶点。

找到一个入度为0的顶点，把它标记为1，并删除与它相连的边，再对剩下的图进行标记，

```

Algorithm Topological_Sorting(G);
Input: G=(V,E) (a directed acyclic graph)
Output: The Label field indicates a topological sorting of G
begin
    Initialize v.Indegree for all vertices; {eg., by DFS}
    G_Label:=0;
    for i:=1 to n do
        if v[i].Indegree=0 then put v[i] in Queue;
    repeat
        remove vertex v from Queue;
        G_Label:=G_Label+1;
        v.Label:=G_Label;
        for all edges (v,w) do
            w.Indegree:=w.Indegree-1;
            if w.Indegree=0 then put w in Queue;
        until Queue is empty
    end

```

复杂性：对变量Indegree进行初始化需要 $O(|V| + |E|)$ 时间，找到一个入度为0的顶点需要常数式时间(访问一个队列)。当v从队列中删除时每一条边(v,w)被访问一次，因此，变量需要被更新的次数正好等于图中边的数目。从而算法的运行时间是 $O(|V| + |E|)$ ，这对于输入规模是线性的。

单源最短路径

问题：给定有向图 $G=(V, E)$ 及一个顶点 v ，寻找到从 v 出发到达 G 中其他各顶点的最短路径。

设图是非循环的，对顶点数目进行归纳。基础情形是平凡的。令 $|V|=n$ ，使用拓扑排序的结论。如果 v 的标记是 k ，则所有标记 $< k$ 的顶点都无需考虑，从 v 出发没有路径到达这些顶点。

以拓扑排序得到的次序作为归纳次序。考虑最后一个顶点，即标记为 n 的顶点 z 。假设已知从 v 出发到达除 z 以外的所有顶点的最短路径。用 $w.SP$ 表示从 v 到 w 的最短路径长度。为了找到 $z.SP$ ，仅需检查那些有边导向 z 的顶点 w 。由于已知到达所有其他顶点的最短路径，则在全部有边到达 z 的 w 中，计算 $w.SP+length(w,z)$ ，取 $z.SP$ 等于最小值。由于 z 是拓扑排序中最后的顶点，在图中不存在其他从 z 出发可以到达的顶点，所以其他路径不受影响。

```

Algorithm Acyclic_Shortest_Paths(G,v,n)
Input: G=(V,E) (a weighted acyclic graph), v (a vertex), and n (the number of
vertices)
Output: For every vertex w in V, w.SP is the length of shortest path
{we assumed that a topological sort has already been performed}
begin
    let z be the vertex labeled n {in the topological order};
    if z!=v then
        Acyclic_Shortest_Paths(G-z,v,n-1);
        for all w such that (w,z) in E do
            if w.SP+length(w,z)<z.SP then
                z.SP:=w.SP+length(w,z);
    else v.SP:=0
end

```

可以使得拓扑顺序和最短路径同时被找到吗？

```

Algorithm Improved_Acyclic_Shortest_Paths(G,V)
Input: G=(V,E) (a weighted acyclic graph), v (a vertex of G)
Output: For every vertex w, w.SP is the length of the shortest path from v to w
begin

```

```

for all vertices w do
    w.SP:=+inf;
Initialize v.indegree for all vertices; {by DFS}
for i:=1 to n do
    if v[i].indegree=0 then put v[i] in Queue;
v.SP:=0;
repeat
    remove vertex w from Queue;
    for all edges (w,z) do
        if w.SP+length(w,z)<z.SP then
            z.SP:=w.SP+length(w,z);
            z.indegree:=z.indegree-1;
            if z.indegree=0 then put z in Queue;
    until Queue is empty
end

```

复杂性：在顶点入度初始化和边的尾部从队列中删除时，每条边都被检查一次。队列的访问时间是常数式时间。每一个顶点只被考虑过一次。所以，最坏情形的运行时间是 $O(|V| + |E|)$ 。

单源最短路径(一般情形)可循环图 Dijkstra算法

拓扑排序的特性：如果 z 是一个标记为 k 的顶点，则(1)不存在从 z 出发到达其他标记 $< k$ 的顶点的路径，并且(2)不存在从标记 $> k$ 的顶点到达 z 的路径。

思路：按照从 v 出发的最短路径长度的顺序来考虑图中顶点。

首先，检查从 v 出发的所有边，令 (v, x) 是它们之间长度最小的边。由于所有的长度都是正数，则从 v 到达 x 的最短路径就是边 (v, x) 。其它从 v 出发的边至少也是同样长度。所以，我们知道了到达 x 的最短路径，这可以作为归纳的基础

选择次靠近 v 的顶点，要考虑的路径是从 v 出发的一条边或者是包含两条边的路径 $((v, x)$ 及从 x 出发的边)，选择其中最小的。

已知与 v 最近的 k 个顶点，以及到达它们的最短路径长度。从 v 到 w 的最短路径只能经过前面的 k 个顶点，比较所有这样的路径，再从中挑出最短的一条。

```

Algorithm Single_Source_Shortest_Paths(G,v);
Input: G=(V,E) (a weighted directed graph), v (the source vertex)
Output: for each vertex w, w.SP is the length of the shortest path from v to w
begin
    for all vertices w do
        w.mark:=false, w.SP:=+inf;
    v.SP=0;
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that w.SP is minimal;
        w.mark:=true;
        for all edges (w,z) such that z is unmarked do
            if w.SP+length(w,z)<z.SP then z.SP:=w.SP+length(w,z)
end

```

把所有未求得最后最短路径的顶点保留在一个堆里，同时用它们目前已知的从 v 到达它们的最短路径长度作为关键字。

复杂性：更新路径长度需要 $O(\log m)$ 次比较，其中 m 是堆的规模。有 $|V|$ 次迭代，导致 $|V|$ 次从堆中的删除。同时，至多有 $|E|$ 次更新(因为每条边至多有一次更新)，导致在堆中有 $O(|E| \log |V|)$ 次比较。所以，运行时间是 $O((|E| + |V|) \log |V|)$ 。

最小代价生成树MCST

问题：给定一个无向连通图 $G=(V, E)$ ，找出 G 的具有最小代价的生成树 T 。

归纳假设1：已知如何为边数 $< m$ 的连通图找到MCST。

具有最小代价的边必然包含在MCST中，把这条边从图中删除，把归纳应用到剩下的图上。

但在删除一条边后的问题与原始问题并不一致。首先，对这条边的选择局限了其他边的选择。其次，在删除了一条边后，图有可能变成不连通了。

归纳假设2：给定一个连通图 $G=(V, E)$ ，已知如何找到 G 的一个 $k (k < |V| - 1)$ 条边的子图 T ， T 是树且是 G 的MCST的一个子图

已经知道 T 是MCST的一部分。因此，在MCST中至少有一条边，把 T 与不属于 T 的顶点相连。

被选择的第一条边是具有最小代价的边， T 被定义为仅有一条边的树。在每一次迭代中，需要找到把 T 与它之外顶点相连的最小代价边。

```
Algorithm MCST(G);
Input: G (a weighted undirected graph)
Output: T (a minimum-cost spanning tree of G)
begin
    Initially T is the empty set;
    for all vertices w do
        w.Mark:=false; {w.Mark is true if w is in T}
        w.Cost:=+inf;
    let (x,y) be a minimum cost edge in G;
    x.Mark:=true;
    for all edges (x,z) do
        z.Edge:=(x,z); {a minimum cost edge from T to z}
        z.Cost:=cost(x,z); {the cost of z.Edge}
    while there exists an unmarked vertex do
        let w be an unmarked vertex such that w.Cost is minimal;
        if w.Cost=+inf then
            print "G is not connected";
            halt
        else
            w.Mark:=true;
            add w.Edge to T;
            {update the costs of unmarked
            vertices connected to w}
            for all edges (w,z) do
                if not z.Mark then
                    if cost(w,z)<z.Cost then
                        z.Edge:=(w,z);
                        z.Cost:=cost(w,z)
    end
```

复杂性：该算法的复杂性等同于单源最短路径算法，最坏情形运行时间是 $O(|V| + |E|) \log |V|$ 。

寻找MCST的算法，是能够找到最优解的贪心法的一个例子

全部最短路径

问题：给定一个(有向或无向)加权图 $G=(V,E)$ ，其权值为非负数，要找到所有顶点对之间的最小长度路径。

归纳方法：对边使用归纳，还是对顶点使用归纳？

增加一条新的边 (u,w) 到图中时：这条边可能是 u 和 w 间的最短路径，也可能有其他的最短路径经过 (u,w) 。在最坏情形，需要检查每一对顶点 v_1 和 v_2 ，看在两者之间是否存在经过 (u,w) 的更短路径。这样要对每一条边做 $O(|V|^2)$ 次检查，这导致最坏运行时间 $O(|E||V|^2)$ 。

增加一个新的顶点 u 到图中：首先需要找到从 u 到所有其它顶点的最短路径长度以及从其它顶点到 u 的最短路径长度。其次必须检查每一对顶点，看在两者之间是否存在一个经过新顶点 u 的更短路径。每增加一个顶点，总共需要 $O(|V|^2)$ 次比较和加法，这将导致一个 $O(|V|^3)$ 的算法。

从1到 $|V|$ 对顶点进行标记。一个从 u 到 w 的路径被称为 k -path，如果除了 u 和 w ，在路径上最大的顶点标记为 k 。特别的，一个0-path是一条边。

归纳假设：已知任意一对顶点之间 k -path的最短路径长度，其中 $k < m$ 。求任意一对顶点之间 m -path的最短路径长度

用 v_m 表示标记为 m 的顶点。任何最短 m -path必须包含 v_m 一次。 u 和 w 之间最短的 m -path，是 u 和 v_m 之间最短的 k -path($k < m$)再加上 v_m 和 w 之间最短的 j -path($j < m$)。

```

Algorithm All_Pairs_Shortest_Paths(Weight)
Input: Weight (an n*n adjacency matrix representing a weighted graph)
Output: At the end, the matrix Weight contains the lengths of the shortest paths
begin
    for m:=1 to n do
        for x:=1 to n do
            for y:=1 to n do
                if weight[x,m]+weight[m,y]<weight[x,y] then
                    weight[x,y]:= weight[x,m]+weight[m,y]
end

```

复杂性：对每一个 m ，算法仅涉及到每一对顶点间的一次加法和一次比较。归纳序列的长度是 $|V|$ ，所以加法(和比较)的数目至多为 $|V|^3$ 。

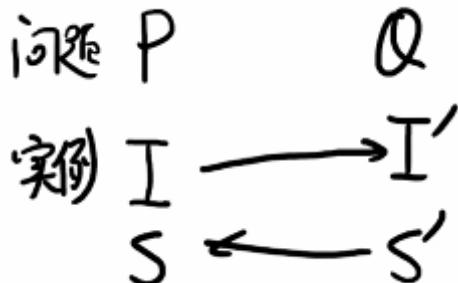
单源最短路径的运行时间是 $O(|E|\log|V|)$ 。如果图是稠密的，即它的边数是 $\Omega(n^2)$ ，则使用该算法要优于为每一个顶点使用单源算法。尽管有可能在 $O(|V|^2)$ 时间内实现单源算法，从而所有顶点对的最短路径存在 $O(|V|^3)$ 的算法，但是对于稠密图，本算法仍然较好，因为它易于实现。如果图是相对稀疏的，通过使用单源算法 $|V|$ 次较好，其运行时间是 $O(|E||V|\log|V|)$ 。

传递闭包

给定一个有向图 $G=(V,E)$ ， G 的传递闭包 $C=(V,F)$ 是一个有向图，满足当且仅当在 G 中存在一条从 v 到 w 的路径，则在 C 中存在一条边 (v,w) 。

问题：给定一个有向图 $G=(V,E)$ ，找到它的传递闭包。

使用归约来求解这个问题，即把传递闭包问题实例转换成已知如何求解的另一问题的实例，然后把后者的解答转换成传递闭包问题的解答。此处归约利用全部最短路径问题。



令 $G'=(V,E')$ 是一个完全有向图(即所有的顶点之间都有双向连接)。对 E' 中每一条边 e , 如果 $e \in E$ 则赋予长度值0, 否则为1。然后对图 G' 求解全部最短路径问题。如果在 G 中从 v 到 w 存在一条路径, 则在 G' 中它的长度值为0, 因为所有 G 中的边在 G' 中长度为0。所以, 在 v 和 w 之间存在一条路径, 当且仅当 v 和 w 的最短路径长度在 G' 中为0。

```
Algorithm Transitive_Closure(A)
Input: A (an n*n adjacency matrix representing a directed graph. A[x,y] is true
       if the edge (x,y) belongs to the graph, and false otherwise. A[x,x] is true for
       all x.)
Output: At the end, the matrix A represents the transitive closure of the graph.
begin
    for m:=1 to n do
        for x:=1 to n do
            for y:=1 to n do
                if A[x,m] and A[m,y] then A[x,y]:=true
end

Algorithm Improved_Transitive_Closure(A)
Input: A (an n*n adjacency matrix representing a directed graph. A[x,y] is true
       if the edge (x,y) belongs to the graph, and false otherwise. A[x,x] is true for
       all x.)
Output: At the end, the matrix A represents the transitive closure of the graph.
begin
    for m:=1 to n do
        for x:=1 to n do
            if A[x,m] then
                for y:=1 to n do
                    if A[m,y] then A[x,y]:=true
end
```

双连通分支

一个无向图是连通的, 如果从每一个顶点到另一顶点都存在一条路径。

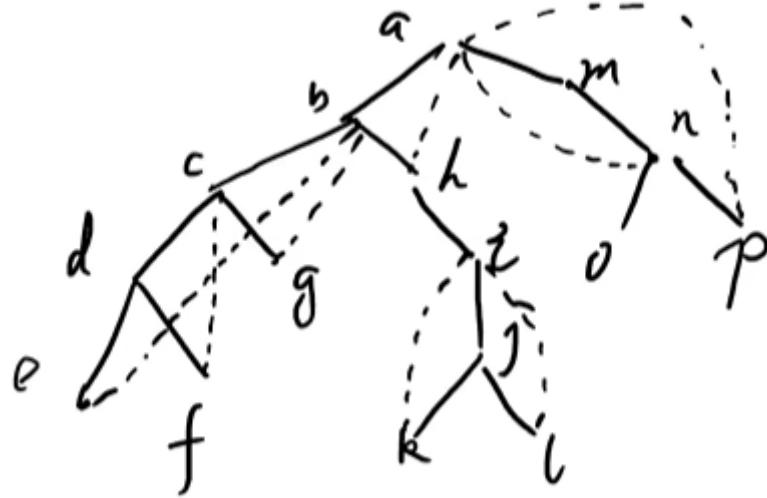
一个无向图是双连通的, 如果从每一顶点到另一顶点至少存在两条不相交的路径。一个无向图被称为 k 连通的, 如果在每两个顶点之间至少存在 k 条不相交的路径

Menger定理: $G=(V,E)$ 是一个无向连通图, u 和 v 是 G 中两个不相邻的顶点。为了使 u 和 v 不连通, 要从 G 中删除的顶点数目至少等于 u,v 之间不相交的路径的最大数

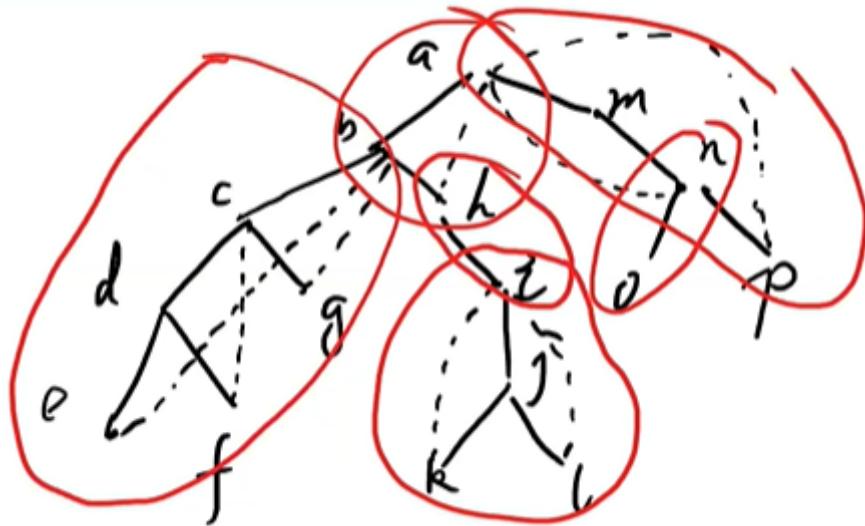
Whitney定理: 一个无向图是 k 连通的, 当且仅当为了使图不连通至少需要删除 k 个顶点。

一个图不是双连通的当且仅当存在一个顶点, 把它删除后即可分离该图。这样的一个顶点被称之为关节点。

一个双连通分支是一个边的最大子集, 其导出子图是双连通的(即不存在一个更大的把它包含在内的子图, 其导出子图是双连通图)。



双连通分支



引理7.9：两条边 e 和 f 属于同一个双连通分支，当且仅当有一条包含着这两条边的闭链。(注意，一个双连通分支可能仅包含一条边，此引理仅适用于至少包含两条边的双连通分支)

引理7.10：每一条边仅属于一个双连通分支。

如何划分

一个具有一条边的连通图是双连通的。

归纳假设：已知如何找到边数 $< m$ 的连通图的双连通分支。

考虑一个具有 m 条边的图，并任意挑一条边 x 。把 x 从图中删除，由归纳假设可以找到双连通分支。

问题：增加 x 后对划分产生什么影响？

双连通树：每一个双连通分支看成一个节点，从任一分支 R 作为树的根开始， R 的儿子是那些与 R 之间具有共同关节点的双连通分支，孙子是那些还没有包含在树中的双连通分支，它们与儿子之间具有共同的关节点，以此类推，即按照广度优先的方式来构造这棵树。

对于双连通树，一条连接来自不同分支顶点的边在树中产生一个闭链，在闭链中所有的节点(对应于分支)必然合并成一个大的分支。

增加归纳假设：已知如何构造双连通树

问题：需要多少时间来找到这个在增加了一个边到双连通树中后产生的闭链？在一棵树中找到一个闭链可能需要遍历整棵树，在最坏情形下需要查找树中所有的边。在树中可能的边数为 $O(|V|)$ ，我们不得不为原始图中的每条边都执行这个步骤。从而，算法可能需要 $O(|V| \times |E|)$ 时间，要避免在每一步都搜索闭链。

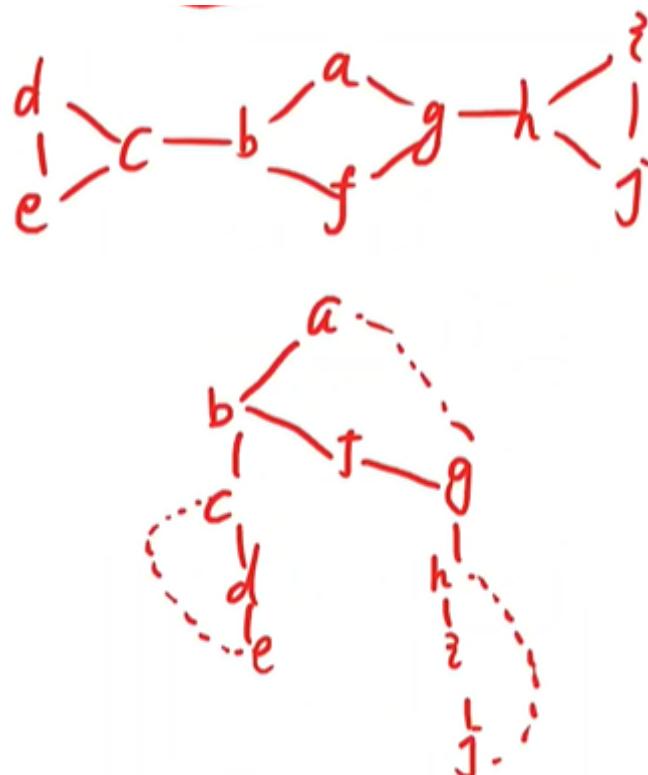
解决方法：仔细选择归纳的次序。考虑DFS

假设顶点v的祖先包括v本身。

为了表示一个顶点经过其后代以及一条非树边所能到达的最高祖先，对于图G的每个顶点v，定义 $\text{High}(v)$ 为从v经过其后代以及一条非树边所能到达的最高顶点。

假定在DFS树中v的儿子是 w_1, w_2, \dots, w_k ，知道所有 w_i 的 $\text{High}(w_i)$ ，可以容易地计算出 $\text{High}(v)$ 为所有 $\text{High}(w_i)$ 以及所有从v发出的后退边中的最高值。

一个顶点v是关节点，当且仅当存在一个v的儿子 w_i ，其满足 $\text{High}(w_i) \geq v$ 。



归纳假设：当用DFS方式访问第k个顶点时，知道如何找到那些已经访问过的，以及位于该顶点之下的顶点的 High 值。

归纳的次序遵照DFS的次序。当到达一个顶点v时，对v所有的儿子(递归)执行DFS，(由归纳)找到它们的 High 值，并根据定义计算 $\text{High}(v)$ 。同时能够判定一个顶点是否是关节点。

特殊情况：根是一个关节点，当且仅当它在DFS树中有多个儿子。

处理 High 值的一个实际方式就是利用DFS数

递减的DFS数：根具有值为 $|V|$ 的DFS数，而随着每次访问一个新顶点其数值递减。也可以使用负数，即赋予根的DFS数为-1，而随着每次访问一个新顶点其数值递减，此方案的优势在于无须事先知道 $|V|$ 的值。

```

Algorithm Biconnected_Components(G, v, n)
Input: G=(V, E) (an undirected connected graph), v (a vertex serving as the root
       of the DFS tree), n (the number of vertices in G)
Output: the biconnected components are marked and the High values are computed
    
```

```

begin
    for every vertex v of G do
        v.DFS_Number:=0;
    {the DFS numbers will also serve to indicate whether or not the
    corresponding vertices have been visited}
    DFS_N:=n;
    {use decreasing DFS numbers}
    BC(v)
end

procedure BC(v);
begin
    *v.DFS_Number:=DFS_N;
    DFS_N:=DFS_N-1;
    insert v into Stack; {Stack is initially empty}
    v.High:=v.DFS_Number; {initial value}
    *for all edges (v,w) do
        insert (v,w) into Stack; {each edge will be inserted twice}
        if w is not the parent of v then
        {
            *if w.DFS_Number=0 then
                *BC(w);
                if w.High<=v.DFS_Number then {v disconnects w from the rest of
the graph}
                    remove all edges and vertices from Stack until v is reached
                    and mark the subgraph they form as a biconnected component;
                    insert v back into Stack; {v is part of w's component and
possibly others}
                    v.High:=max(v.High, w.High);
                else {(v,w) is a back edge or a forward edge}
                    v.High:=max(v.High, w.DFS_Number)
            }
        end
    end

```

复杂性：除了关于DFS的操作之外，每个顶点的额外工作量是常数。因此，算法的运行时间是 $O(|V|+|E|)$ 。需要的空间也是 $O(|V|+|E|)$ ，因为所有分支必须在遍历的过程中被记住。

强连通分支

一个有向图，如果每一对顶点v和w，从v到w有条路径且从w到v也有条路径，则称是强连通的，即从任一顶点开始可以到达任一其他顶点。

一个强连通分支是一个最大的顶点子集，其导出子图是强连通的(即不存在另一个包含它且能导出一个强连通图的子图)。

引理 7.11：两个顶点属于同一个强连通分支，当且仅当存在一个包含它们的回路。(一个回路是一个封闭的有向路径，但并不一定是简单的，即可以包含一个顶点多次。而一个闭链则是一条简单回路。)

引理 7.12：每个顶点仅属于一个强连通分支。

强连通分支中的所有顶点必定属于DFS树中的一个连通子树

对于一个给定的分支，考察它在树中的最高顶点，称这个顶点为分支的根，是分支中第一个被DFS访问的顶点。

根类似于关节点，从根或其后代出发不存在到达图其它分支的路径

可以用类似于判断关节点的方法判定根

如何划分

令 r 是第一个完全被DFS访问的分支的根。通常在DFS的图形中，它是最左边最低的一个分支。这个分支必然包含 r 在树中的全部后代(不可能有任何一个后代可能属于某个更小的分支，因为分支遍历已经被完成)。

如果在DFS期间，可以判定 r 是第一个根，则可以判定这个分支并把它从图中删除，然后由归纳继续进行下去。

作为某个分支的根的顶点 r ，不可能有任何后向边从 r 的后代指向一个比 r 高的顶点，这样的后向边与这个更高的顶点会构成一个循环，意味着这个更高的顶点与 r 属于同一个分支。可以像双连通分支那样确定是否存在这样的后向边，即利用High值。

交叉边的影响。

一旦找到第一个根就可以找到第一个强连通分支，它包含DFS树的根以及全部的后代，然后可以从图中删除这个分支。现在得到了一个更小的图，剩余的事情就可以通过归纳完成了

High值的定义是动态的。由于删掉了指向刚被发现的分支的边，所以它们在以后的High值计算中不起作用。这有别于双连通分支中High值的“静态”定义，其不依赖于以前的任何分支。

不需要真正删除顶点或者边，可以简单地标记每一个被发现的分支中的顶点，以后忽略那些指向已标记的顶点的边。

```
Algorithm Strongly_Connected_Components(G, v, n)
Input: G=(V, E) (a directed graph), v (a vertex serving as the root of the DFS
tree), and n (the number of vertices in G)
Output: marking the strongly connected components, and computing the High
values.

begin
    for every vertex v of G do
        v.DFS_Number:=0;
        v.Component:=0;
        Current_Component:=0;
        DFS_N:=n;
        while there exists a vertex v such that v.DFS_Number=0 do
            SCC(v)
    end

    procedure SCC(v)
    begin
        v.DFS_Number:=DFS_N;
        DFS_N:=DFS_N-1;
        insert v into STACK;
        v.High:=v.DFS_Number;
        for all edges (v,w) do
            if w.DFS_Number=0 then
                SCC(w);
                v.High=max(v.High,w.High);
            else
                if w.DFS_Number>v.DFS_Number and w.Component=0 then
                    v.High:=max(v.High,w.DFS_Number); { (v,w) is a cross edge or a
back edge}
                if v.High=v.DFS_Number then
                    Current_Component:=Current_Component+1;
                    repeat {mark the vertices of the new component}
                        remove x from the top of STACK;
                end
            end
        end
    end
end
```

```

x.Component:=Current_Component;
until x=v
end

```

复杂性：该算法类似于双连通分支算法，复杂性相同，时间和空间复杂性都是 $O(|V|+|E|)$ 。

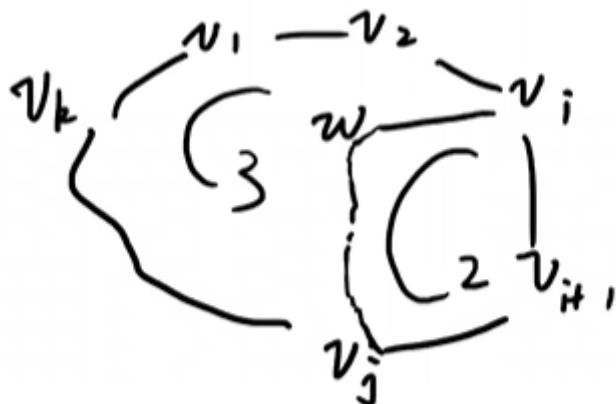
利用图分解的例子(1)

问题：给定一个连通无向图 $G=(V,E)$ ，判定它是否存在一个偶数长度的闭链。

一个闭链必定包含在一个双连通分支中。因此，可以把图划分成双连通分支，然后分别考察每一个分支。

定理 7.13：边数多于一的，并且不只有一个奇数长度闭链的双连通图，必然有一个长度为偶数的闭链。

若图是双连通的，边数 > 1
 存在闭链 $C_1 = v_1 \dots v_k v_1$
 $k \geq 3$ ✓
 假如如果图中所有边都包含在 C_1 中
 存在边 (v_i, w) w 不在 C_1 中
 图双连通 $\therefore (v_i w) (v_i v_{i+1})$
 存在于闭链 C_1 中



$$C_1 = P + Q$$

$$\textcircled{C}_2 = Q + R$$

$$C_3 = P + \textcircled{R} \text{ 保}$$

三个中必有一个偶数

利用图分解的例子(2)

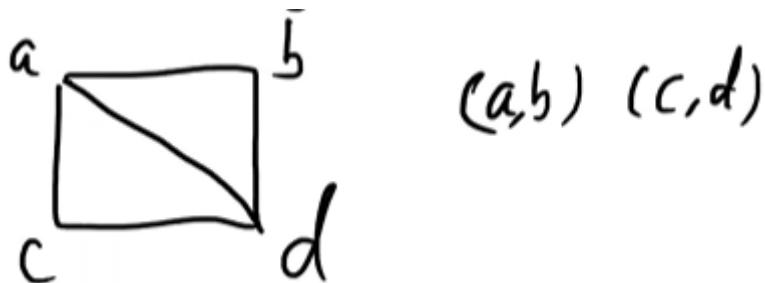
问题：给定一个有向图 $G=(V,E)$ ，判定是否包含一个奇数长度的(有向)闭链。

一个闭链必定包含在一个强连通分支中，所以可以同样假定图是强连通的。

从任一顶点 r 开始执行 DFS，并且对顶点做偶数或者奇数的标记。把 r 标记为偶数，然后对于每一条边 (v,w) ，对 v 和 w 做各自相反的标记。由于 r 可以从任意顶点到达，则有一个奇数长度闭链当且仅当试图对某个顶点进行标记时发现它已经被做过相反的标记了。

匹配

给定一个无向图 $G=(V,E)$ ，匹配是边的一个集合，其中任意两条边没有公共的顶点。



同时是完美，最大，极大

a, d 是极大匹配

一个与匹配中任何边都不相关联的顶点称为无匹配的，或称这个顶点不属于该匹配。

一个完美匹配是所有顶点都有匹配的匹配。

一个最大匹配是具有最大边数的匹配。

一个极大匹配是不可能再增加边数的匹配。最大匹配一定是极大匹配

非常稠密图中的完美匹配

令 $G=(V,E)$ 是一个无向图，其 $|V|=2n$ 并且每个顶点的度至少为 n 。在图G中考虑一个有 m 条边的匹配 M , $m < n$ 。先检查所有不在 M 中的边，看是否其中的一条可以加入到 M 中。如果找到这样的一条边则完成。否则， M 就是个极大匹配。由于 M 不是完美的，至少有两个不相邻的顶点 v_1 和 v_2 不属于 M 。这两个顶点至少有 $2n$ 条各不相同的边指向被 M 覆盖的顶点，如果不这样的话，则有一条边可以加入 M 中。由于 M 的边数 $< n$ 且从 v_1 和 v_2 出发有 $2n$ 条边与它相连，所以至少有一条属于 M 的边，设为 (u_1, u_2) ，与来自 v_1 和 v_2 的三条边相连。不失一般性，假定这三条边是 (u_1, v_1) , (u_1, v_2) 和 (u_2, v_1) ，通过从 M 中删除边 (u_1, u_2) ，并增加两条边 (u_1, v_2) 和 (u_2, v_1) ，可得到一个更大的匹配。

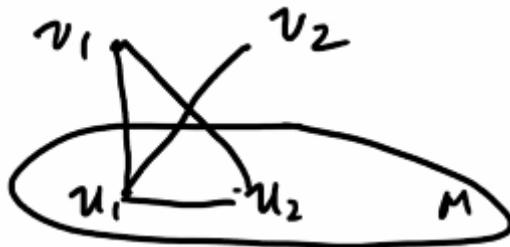
1. 先求极大匹配 M
2. 再去求完美匹配

$$|M|=m < n$$



$$2n > 2m$$

存在下面情况



可以用两边换掉一边得到更大匹配

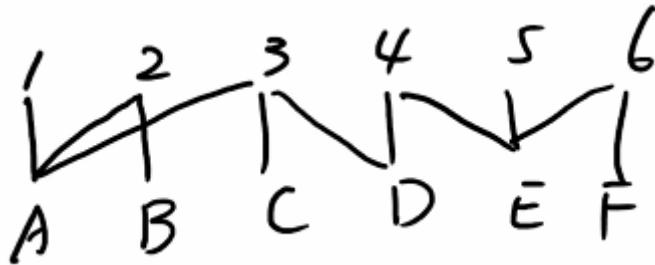
偶图匹配

令 $G=(V,E,U)$ 是一个偶图，其中 V 和 U 是两个不相交的顶点集合， E 是连接 V 和 U 之间顶点的边集合。

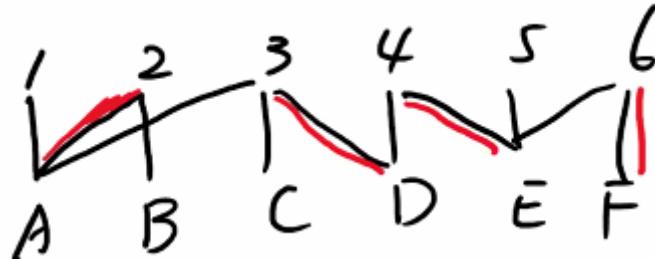
问题：在偶图中找到一个最大匹配。

贪心法：先匹配度数小的顶点，希望后面其它的顶点要比它更容易找到未被匹配过的点。

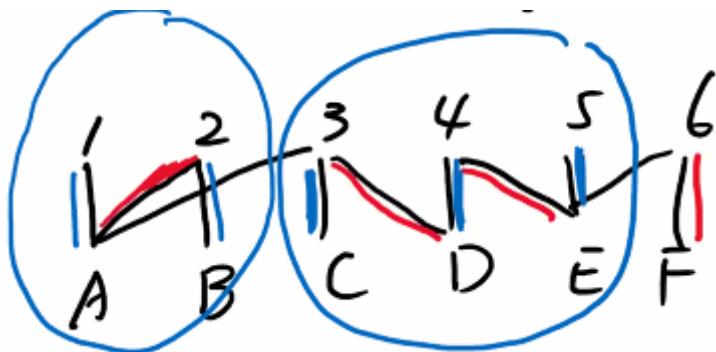
从一个非匹配顶点 v 开始，并尝试为它找到一个配对。如果已经有了一个极大匹配，则 v 的所有邻居都已经被匹配了，所以必须尝试打破这个匹配，尝试用 $k+1$ 条边来替换 k 条边。



如果一开始找的极大匹配如下



对于其中的几组可以进行替换



发现红边蓝边相连可以构成路径。

对于一个给定匹配 M ，一个交互路径 P 是一个从 V 中顶点 v 出发到达 U 中顶点 u 的路径，而 v 和 u 都没有在 M 中被匹配，且 P 中的边交错出现在 $E-M$ 和 M 中。这个交互路径可以用来改进匹配，用 P 中不在 M 里的边来替换 M 里的边，则得到另一个多一条边数的匹配。

1-A(E-M) A-2(M) 2-B(E-M)

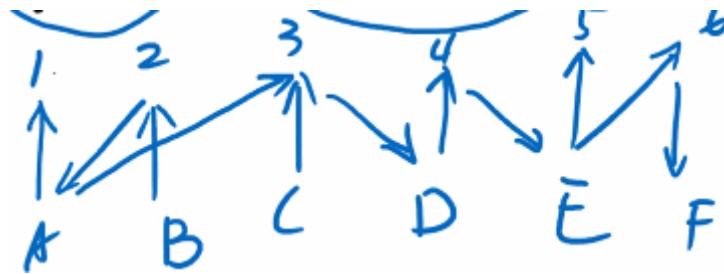
$E-M$ 与 M 交错， $E-M$ 比 M 多一条

交互路径定理：一个匹配是最大的，当且仅当它不存在交互路径。

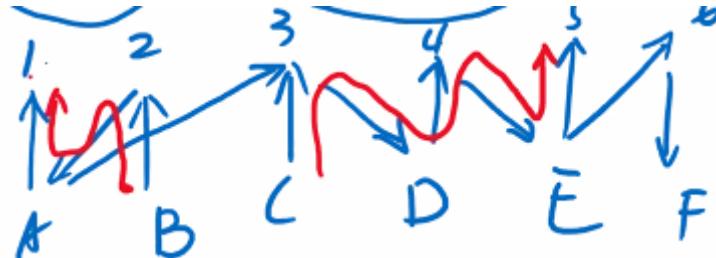
开始利用贪心算法把尽可能多的边加入到匹配中，一直到得到一个极大匹配为止。然后搜索一条交互路径，并相应地修改匹配，一直到找不到交互路径为止，最后得到的匹配就是最大的。由于每一个交互路径按照多增加一条边来扩展一个匹配，而在任何匹配中至多有 $n/2$ 条边(n 是顶点数)，所以迭代次数至多为 $n/2$ 。

如何找到交互路径：通过设定 M 中边的方向是从 U 到 V ，不在 M 中边的方向是从 V 到 U ，把无向图 G 转换成有向图 G' 。一个交互路径恰好对应于一个从 V 中非匹配顶点到 U 中非匹配顶点的有向路径。这样一条有向路径可以通过任意的图搜索过程得到，例如DFS。搜索的复杂性是 $O(|V|+|E|)$ ，因此算法的复杂性是 $O(|V|(|V|+|E|))$

转换为有向图



即找到从下方出发的到达上方的有向路径，可以通过DFS来做



改进——每次产生多个路径

在 G' 中从 V 中未被匹配顶点集开始执行BFS，逐层进行，一直到在 U 中找到未被匹配的顶点。然后，通过BFS导出的图，抽取出 G' 中顶点不相交路径的极大集合(这就是 G 中的交互路径)。找到任何一个路径后，删除它的顶点，然后找另一条路径，再删除顶点，一直下去。为了在搜索中最大化匹配的边数(每一个顶点不相交路径为匹配增加一条边)，选择一个极大集合。最后，利用交互路径集合修改匹配。这个过程一直重复到找不到更多的交互路径。

复杂性：改进过的算法在最坏情形的迭代次数是 $O(\sqrt{V_1})$ 。

哈密尔顿回路

问题：给定图 $G=(V,E)$ ，在 G 中寻找一个简单回路，其包含 V 中每个顶点一次且仅一次。

此回路称为哈密尔顿回路，包含这样一个回路的图被称为哈密尔顿图。

eg. 环和完全图

反向归纳

思想：利用一个无限集 S 作为归纳的基础，证明定理 $P(n)$ 对于所有属于 S 中的 n 值成立。然后，“后退”证明 $P(n)$ 成立蕴含着 $P(n-1)$ 成立。

常用算法1：仅为规模是2的指数的输入设计算法。

常用算法2：当可能的元素数目是有界时，基础情形可以是具有最大元素数目的情况。

在哈密尔顿算法中，完全图作为归纳基础

在非常稠密的图中找哈密尔顿回路

问题：给定连通无向图 $G=(V,E)$ ，其顶点数 $n \geq 3$ ，且每一对非邻接顶点 v 和 w 满足 $d(v) + d(w) \geq n$ ，在 G 中寻找一个哈密尔顿回路。（由Ore定理，图 G 存在哈密尔顿回路）

基础情形是一个完全图，至少有3个顶点的完全图包含一个哈密尔顿回路，只要按任意次序将全部顶点连接成回路即可。

归纳假设：已知如何在满足给定条件，且边数 $\geq m$ 的图中找到一个哈密尔顿回路。

$G=(V,E)$ 是满足给定条件且边数 $=m-1$ 。考虑 G 中任意一对非邻接顶点 v 和 w ，构造图 G' ，其中除了 v 和 w 是邻接之外，其它都与 G 相同。

由归纳假设在 G' 中找到哈密尔顿回路 $x_1, x_2, \dots, x_n, x_1$ 。如果边 (v,w) 不包含在这个回路中则完成，否则设 $v=x_1$ 且 $w=x_n$ 。由 G 中给定的条件 $d(v)+d(w) \geq n$ ， G 中所有从 v 和 w 发出的边至少有 n 条。但是 G 还有 $n-2$ 个其它顶点，所以在回路中存在两个邻接顶点 x_i 和 x_{i+1} ，其中 v 与 x_{i+1} 连接而 w 与 x_i 连接。利用边 (v,x_{i+1}) 和 (w,x_i) 可以得到一个不经过 (v,w) 的新哈密尔顿回路 $v(=x_1), x_{i+1}, x_{i+2}, \dots, w(=x_n), x_i, x_{i-1}, \dots, v$

实现：1) 从完全图开始，每一步取代一条边；2) 对于图 G ，找到一个较大的路径(如通过DFS)，并增加一些(不在 G 中的)边(最坏情况下增加 $n-1$ 条边)，从而构成哈密尔顿回路。从 G' 开始，一直到为 G 找到一个哈密尔顿回路为止。

取代一条边的步骤数是 $O(n)$ ，有 $O(n)$ 条边被取代，所以算法的运行时间是 $O(n^2)$ 。

57

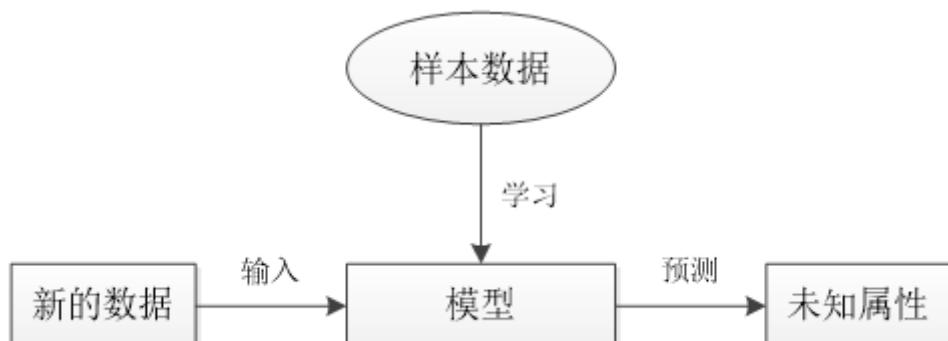


Ch 07_2 数学归纳法，树和机器学习

学习

学习是人类以及各种动物与生俱来的基本能力，自从人们试图在计算机上表现人类智能之日起，学习就成为研究的主要课题。

H.Simon曾于1983年给出了一个关于学习的哲学式说明：如果一个系统能够通过执行某种过程而改进其性能，这就是学习。以常见的有监督学习为例，机器学习可以理解为针对所给定的样本集 $\{(x_i, y_i) : i=1, 2, \dots, n\}$ ，该样本集来自实际问题 $y=F(x)$ （称为自然模型）的独立同分布采样，我们需要设计算法以得到函数 $y=f(x)$ ，使得它对自然模型在一定的统计指标下为真，即 $f(x)$ 是自然模型的一个近似模型



从IF...THEN开始

天热决定如何降温时，收集了一些以前的样例：

温度	是否开空调
29	否
31	是
33	是
26	否
32	是

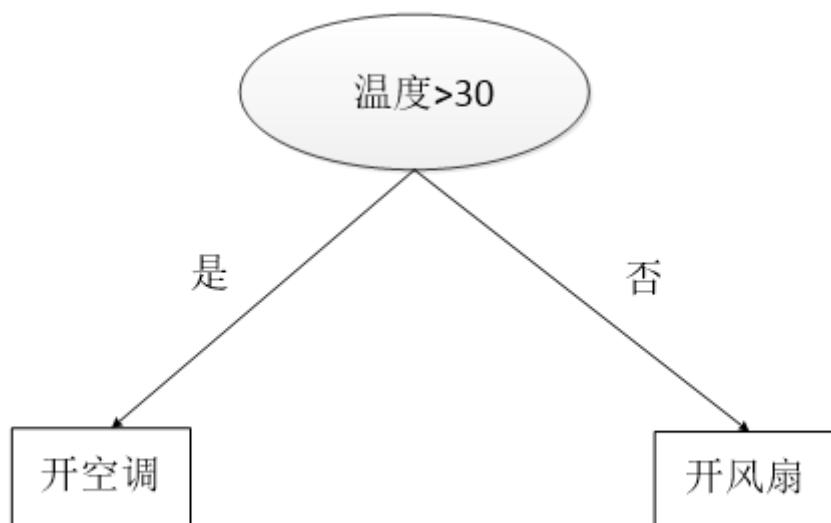
观察分析这些数据，可以很容易得到这样的决策：

IF 温度>30

THEN 开空调

ELSE 开风扇

图形的方式来描述



这就是最简单的决策树，根节点表示在一个属性上的测试或判断，每个分枝代表一个测试输出，而每个树叶节点存放一个相应的决策。

另一个例子

```

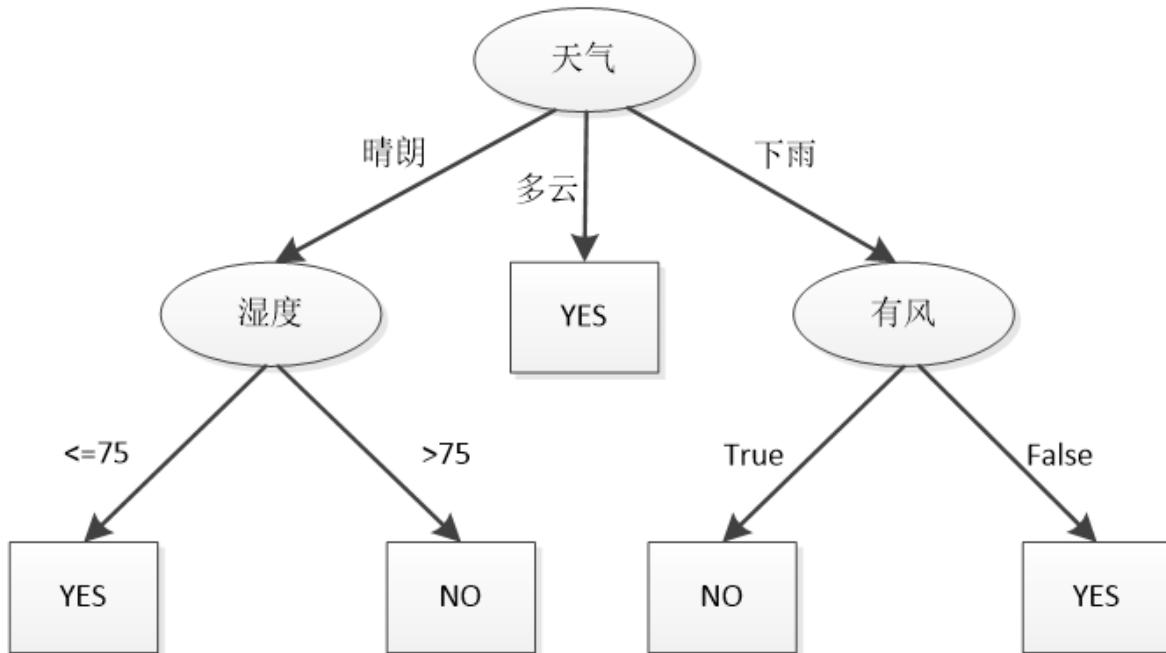
IF 天气晴朗 THEN
  IF 湿度<=75
    THEN 打球
    ELSE 不去打球
ELSEIF 天气多云 THEN 去打球
ELSEIF 下雨 THEN
  IF 有风
    THEN 不去打球
    ELSE 去打球

```

这是一条能够帮助人们根据天气情况决定是否去打高尔夫球的决策规则，与前面天热决定如何降温的规则相比要复杂多了，通过将多个IF...THEN复合嵌套，在增加复杂性的同时我们获得了一条相当“聪明”的规则，可以处理多种复杂情况。

但是比较难理解。

将此规则用另一种容易阅读并理解的形式表示出来



决策树

决策树可以看成是一个IF...THEN规则的集合，是一种类似流程图的树结构，其中每个内部节点（非树叶节点）表示在一个属性上的测试，每个分枝代表一个测试输出，而每个树叶节点存放一个类标号。

一旦建立了决策树，对于一个需要决策的输入，跟踪一条从根节点到叶节点的路径，该叶节点就存放着该输入的预测（或者分类、决定等）。

因此决策树可以看作由条件IF（内部节点）和满足条件时对应的规则THEN（边）组成的。

如何得到决策树？

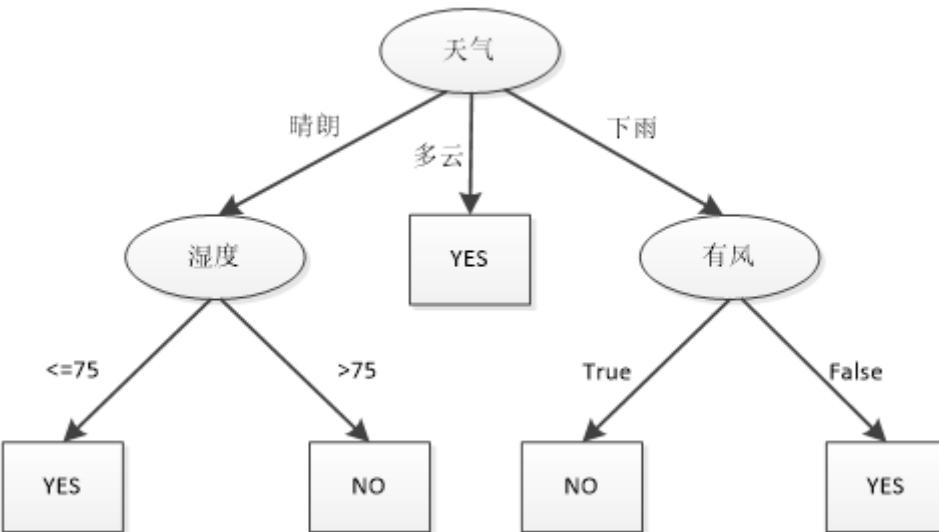
通过学习，通过学习事先收集的实例，就可以得到决策树。

Day	Outlook	Temperature	Humidity	Windy	Play?
1	Sunny	85	85	False	No
2	Sunny	80	90	True	No
3	Overcast	83	78	False	Yes
4	Rainy	70	96	False	Yes
5	Rainy	68	80	False	Yes
6	Rainy	65	70	True	No
7	Overcast	64	65	True	Yes
8	Sunny	72	95	False	No
9	Sunny	69	70	False	Yes
10	Rainy	75	80	False	Yes
11	Sunny	75	70	True	Yes
12	Overcast	72	90	True	Yes
13	Overcast	81	75	False	Yes
14	Rainy	71	80	True	No

怎样从数据集得到决策树？

数据集本身有很多属性，我们怎么知道首先要对哪个属性进行判断，接下来要对哪个属性进行判断？

在图中，我们怎么知道第一个要测试的属性是Outlook，而不是Windy？而且，学习实例中包含了温度信息，但是决策树中却没有涉及到，这又是为什么呢？



常见的决策树学习算法

常见的决策树学习算法有ID3、C4.5、CART等

这些不同的学习算法主要差别在于它们选择特征的依据不同，而决策树的生成过程都是一样的，都是根据当前环境对特征进行贪婪的选择，从而将学习样本集分成不同的子集，得到规模小一些的分类问题。

ID3算法

ID3算法也采用数学归纳法的思想，将原始问题进行分解，降低问题规模。

最容易解决的问题是所有训练样例都属于同一类的情况，这时不需要做任何测试判断，直接就可以输出结果了

如果训练样例不属于同一类，则通过自顶向下构造决策树来进行学习。第一个需要解决的问题是树的根节点测试哪一个属性？

为了回答这个问题，需要利用统计测试来求出每一个实例属性单独分类训练样例的能力，分类能力最好的属性将被选作根节点的测试属性。然后为根节点属性的每个可能值产生一个分支，并将训练样例分配到适当的分支之下。

通过以上操作，原始的决策树构造问题就被分解成数个小规模的决策树构造问题。重复前面的过程，用每个分支节点关联的训练样例来选取在该分支节点测试的属性，直至每个分支下的训练样例属于同一类为止。

算法：ID3（学习布尔函数）

输入：学习样本集合Example，样例属性集

输出：决策树

1. 创建树的根节点
2. 如果样例都为正，则返回标签为正的单节点树
3. 如果样例都为负，则返回标签为负的单节点树
4. 求出属性中对样本分类能力最好的属性A
5. 对于属性A的每一个可能值vi
 - 1) 在根节点下增加分支对应A=vi的测试结果
 - 2) 设Example(i)为Example中满足属性A的值为vi的子集
 - 3) 递归调用ID3(Example(i), 样本属性集-{A})

如何求对样本分类能力最好的属性

ID3算法使用信息增益来衡量给定的属性区分训练样例的能力，从而在候选属性中选择

熵是表示随机变量不确定性的度量。设X是一个取有限个值的离散随机变量，其概率分布为：

$$Prob(X = x_i) = p_i$$

则随机变量X的熵定义为： $H(X) = - \sum p_i \log p_i$

熵只依赖X的分布，和X的取值没有关系。熵是用来度量不确定性，当熵越大，意味着随机变量X的不确定性越大，反之越小，相应的在机器学习分类中，熵越大即表示这个类别的不确定性更大，反之越小。

条件熵表示在已知随机变量X的条件下随机变量Y的不确定性度量。设有两个随机变量X和Y，在随机变量X给定的条件下随机变量Y的条件熵H(Y|X)，定义为X给定条件下Y的条件概率分布的熵对X的数学期望：
$$H(Y|X) = \sum Prob(X = x_i) H(Y|X = x_i)$$

信息增益表示得知特征X的信息而使得类Y的信息的不确定性减少的程度。特征A对训练数据集D的信息增益g(D, A)，定义为集合D的熵H(D)与特征A给定条件下D的条件熵H(D|A)之差，即：

$$g(D, A) = H(D) - H(D|A)$$

信息增益大的特征具有更强的分类能力。

14个样本，其中9个为yes，5个为no，因此该训练样本集合S的熵为

$$H(S) = -9/14 \log(9/14) - 5/14 \log(5/14) = 0.940$$

考虑风力windy的信息增益

$$H(S|Windy)$$

$$= \text{Prob}(Windy = \text{True})H(S | Windy = \text{True}) + \text{Prob}(Windy = \text{False})H(S | Windy = \text{False}) \\ = 6/14 * (-3/6\log(3/6) - 3/6\log(3/6) + 8/14 * (-2/8\log(2/8) - 6/8\log(6/8)) = 0.892$$

因此 $g(S, Windy) = H(S) - H(S | Windy) = 0.048$

类似的，我们可以求出其它三个属性的信息增益分别为：

$$g(S, Outlook) = 0.246$$

$$g(S, Humidity) = 0.151$$

$$g(S, Temperature) = 0.029$$

由于Outlook的信息增益最大，即属性outlook在训练样本上提供了对决策目标的最佳预测，因此outlook被选作根节点的决策属性

当outlook被选作根节点的决策属性后，要为其每一个可能取值（sunny, overcast, rainy）在根节点下创建分支，所提供的学习样本根据其outlook属性的取值情况分为3组供后续决策树的构造。

outlook属性取值为overcast的分支，由于其分配到的样本其决策值都为yes，所以该分支下面的节点就是一个值为yes的叶节点

其余两个分支，由于分配到的样本决策值同时存在yes和no，因此需要重复刚才的过程，选择新的属性来分割学习样本，并且仅使用与这个分支相关的样本，如outlook属性取值为sunny的分支，继续学习时只使用outlook属性为sunny的5个样本。

学习过程结束的标准

对于一条从根节点开始的路径，如果所有属性都在这条路径中测试过，则该路径不再扩展。

对于一个测试节点而言，如果其某个分支分配到的样本其决策值一致，则该分支下面的节点为叶节点。

缺点

在上述ID3算法中，决策树的学习是根据信息增益来进行属性的选择的

但是会偏向于具有大量值的属性。

在训练集中，某个属性所取的不同值的个数越多，那么越有可能拿它来作为分裂属性。

例如在上述的例子中增加一个日期属性，每个样本的日期都不同，则在所有属性中，日期属性具有最大的信息增益，单独的日期就可以完全预测训练数据的目标属性，于是日期属性就会被选中作为根节点的决策属性并形成一棵深度为1但是非常宽的树，这棵树可以百分之百地正确分类训练数据，但对于不在训练集中的数据，由于日期不一致，因此无法进行分类，因而不是一个好的预测器。

改进

C4.5采用了信息增益率

-信息增益率 $g_R(D, A)$ 定义为其信息增益 $g(D, A)$ 与训练数据集 D 关于特征 A 的值的熵 $H_A(D)$ 之比，即

$$g_R(D, A) = \frac{g(D, A)}{H_A(D)}$$

-其中分裂信息为

$$-H_A(D) = - \sum_{i=1}^n \frac{|D_i|}{|D|} \log \frac{|D_i|}{|D|}$$

-在这些公式中， D_1, D_2, \dots, D_n 是根据属性A对样本集分割形成的n个样本子集，每个子集中的样本属性A取值相同。

CART算法则以基尼指数 (gini index) 做为属性选择的依据。

-在分类问题中，假设有K个类，样本属于第k类的概率为 p_k ，则概率分布的基尼指数定义为：

$$-Gini(p) = \sum p_k(1 - p_k) = 1 - \sum p_k^2$$

-对于二分类问题，若样本点属于第一个类的概率是p，则概率分布的基尼指数为 $Gini(p)=2p(1-p)$

-对于给定的样本集合D，其基尼指数为：

$$-Gini(D) = 1 - \sum \left(\frac{|D_k|}{|D|} \right)^2$$

-这里， D_k 是D中属于第k类的样本子集，k是类的个数。

-如果样本集合D根据特征A是否取到某一可能值a被分割成 D_1 和 D_2 两部分，则在特征A的条件下，集合D的基尼指数定义为：

$$-Gini(D, A) = \frac{|D_1|}{|D|} * Gini(D_1) + \frac{|D_2|}{|D|} * Gini(D_2)$$

-基尼指数 $Gini(D)$ 表示集合D的不确定性，基尼指数越大，样本集合的不确定性也就越大，这一点与熵相似。CART用作分类树时采用基尼指数最小化原则，进行特征选择，递归地生成决策树。

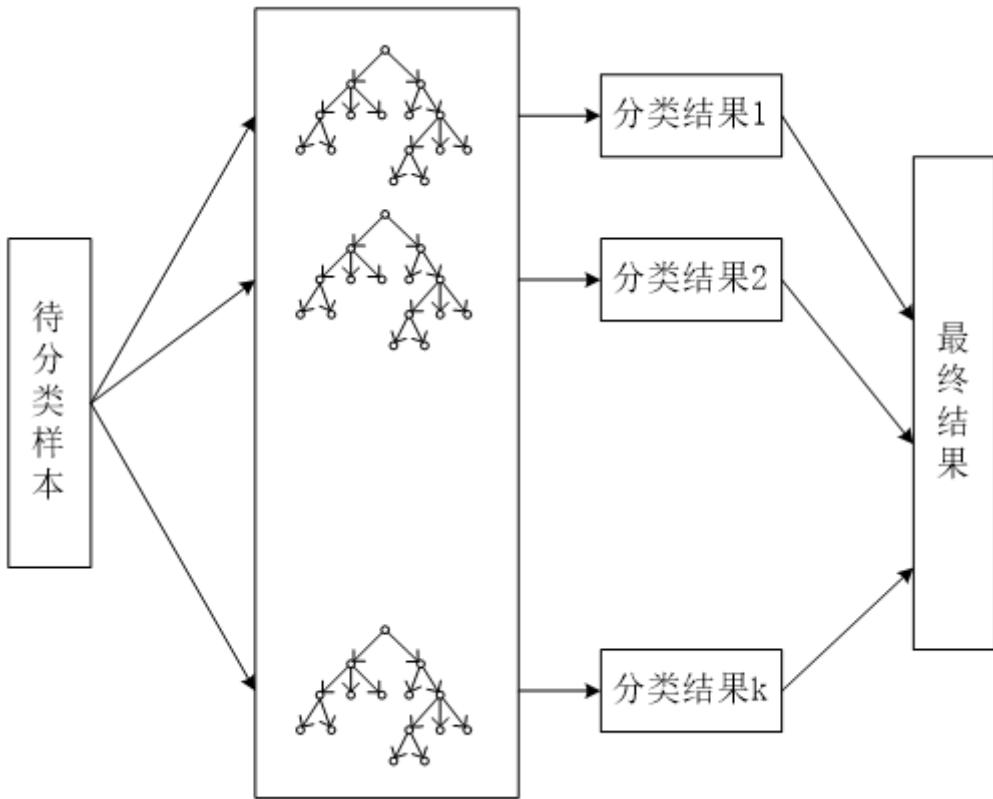
小结

不管是采用哪一种学习算法，其基本思路都是利用统计测试来计算每一个实例属性单独分类训练样例的能力，分类能力最好的属性将被选作根节点的测试属性。然后为根节点属性的每个可能值产生一个分支，并将训练样例分配到适当的分支之下。通过这样的操作，原始的决策树构造问题就被分解成数个小规模的决策树构造问题。重复前面的过程，用每个分支节点关联的训练样例来选取在该分支节点测试的属性，直至每个分支下的训练样例属于同一类为止

性能良好的决策树是一棵与学习样本矛盾较小的决策树，同时具有良好的泛化能力，即好的决策树不仅对学习样本有着很好的分类效果，对于新的数据也有着较低的错误率。

随机森林

随机森林是通过集成学习的思想将多棵树集成的一种算法，它的基本单元是决策树



集成分类器性能与个体关系

	样本1	样本2	样本3		样本1	样本2	样本3		样本1	样本2	样本3
分类器1	F	T	T	分类器1	T	T	F	分类器1	T	F	F
分类器2	T	F	T	分类器2	T	T	F	分类器2	F	T	F
分类器3	T	T	F	分类器3	T	T	F	分类器3	F	F	T
集成结果	T	T	T	集成结果	T	T	F	集成结果	F	F	F

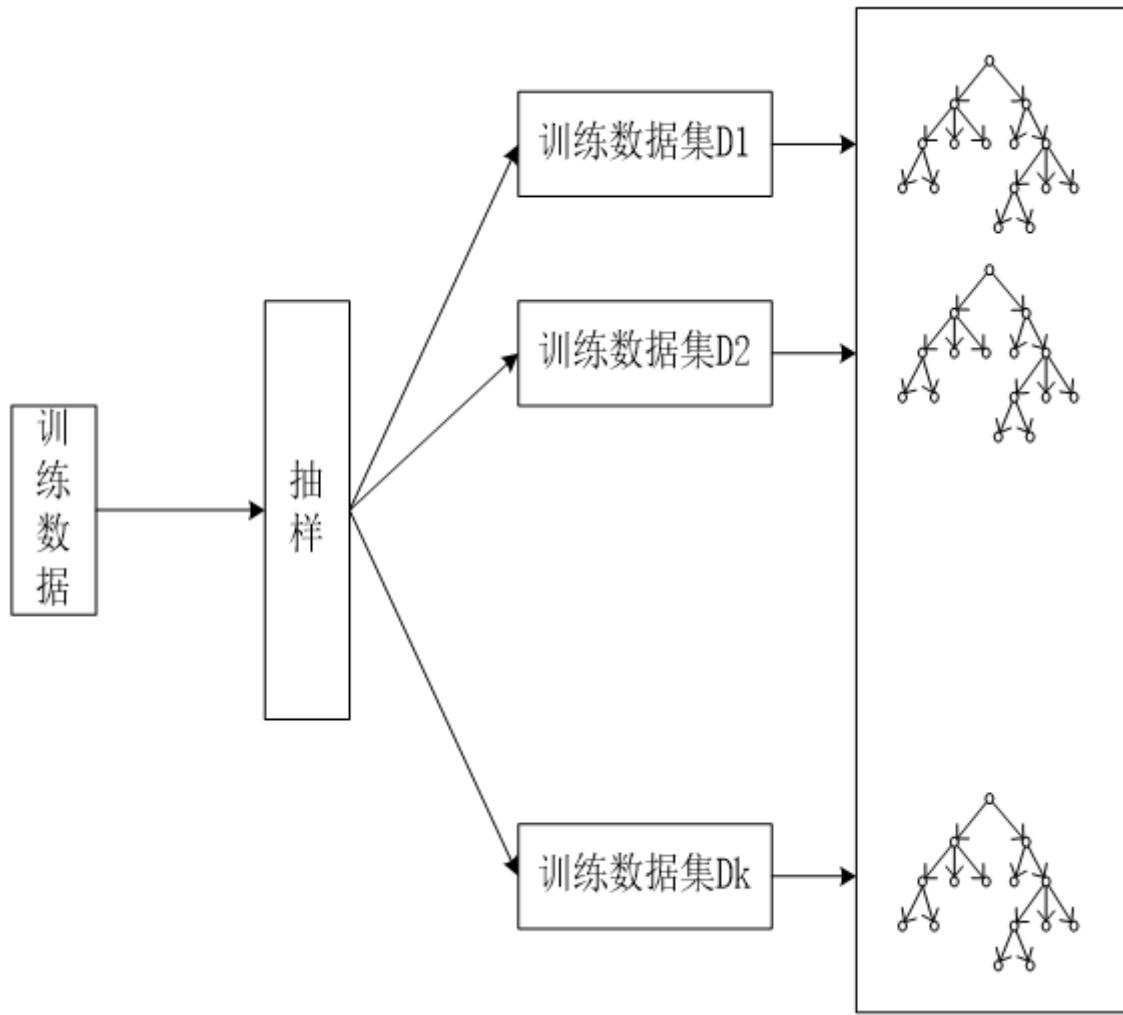
结论：要获得好的集成结果，一方面每个个体分类器都要有一定的准确率，同时每个分类器的表现要尽量有差异性

常用的两类方法：bagging和boosting

Bagging

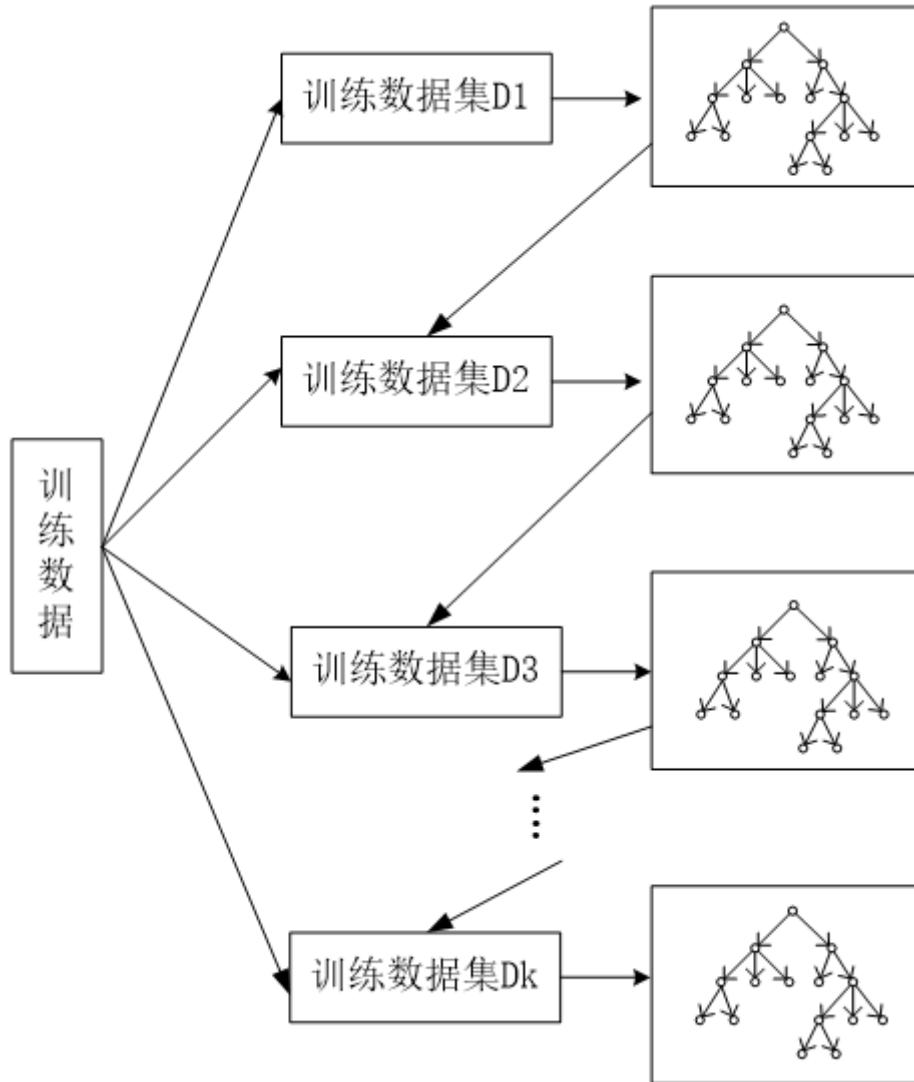
对于初始给定的训练集，每次从中随机取出n个训练样本组成每轮的训练集，某个初始训练样本在某轮训练集中可以出现多次或根本不出现，训练之后可得到一个决策树序列 h_1, \dots, h_n 构成最终的随机森林H，H对分类问题采用投票方式决定最后的结果

决策树在进行节点分裂时，不是所有的属性都参与属性指标的计算，而是随机地选择某几个属性参与比较，参与的属性个数就称之为随机特征变量。随机特征变量是为了使每棵决策树之间的相关性减少，同时提升每棵决策树的分类精度，从而提升整个森林的性能。



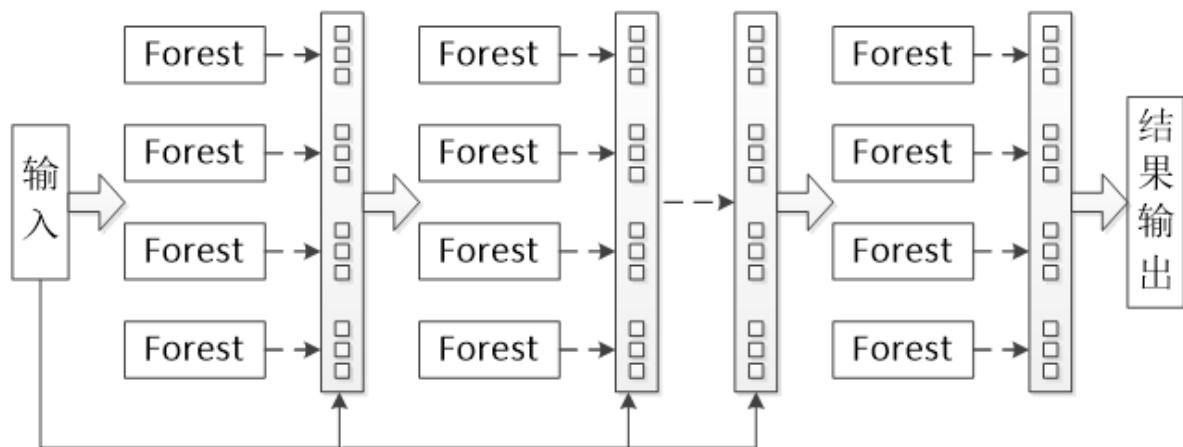
Boosting

boosting的训练则是串行进行的，第 k 个分类器训练时关注对前 $k-1$ 分类器中错分的样本，即不是随机取，而是加大取这些样本的概率。初始化时对每一个训练样本赋相等的权重 $1 / n$ ，然后用该学算法对训练集训练 t 轮，每次训练后，对训练失败的训练样本赋以较大的权重，也就是让学习算法在后续的学习中集中对比较难的训练样本进行学习，从而得到一个决策树序列 h_1, \dots, h_n 构成最终的随机森林 H ，其中每个 h_i 也有一定的权重，预测效果好的决策树权重较大，反之较小， H 对分类问题采用有权重的投票方式。



深度随机森林

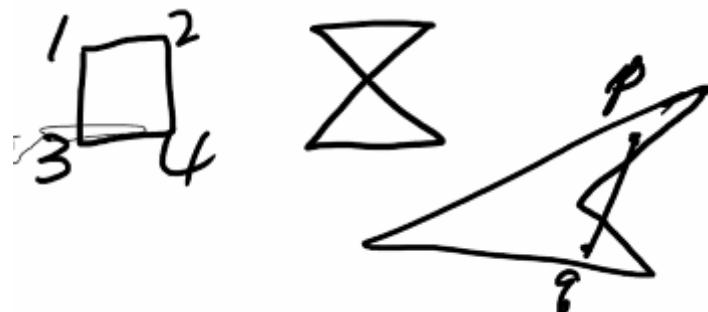
深度随机森林通过对树组成的森林来集成并前后串联起来达到表征学习的效果。它的学习能力可以通过对高维输入数据的多粒度扫描而进行加强，串联的层数也可以通过自适应的决定从而使得模型复杂度不需要成为一个自定义的超参数，而是一个根据数据情况而自动设定的参数。深度随机森林采用了深度网络的串联结构，从前层输入数据，输出结果及源输入作为下层的输入。



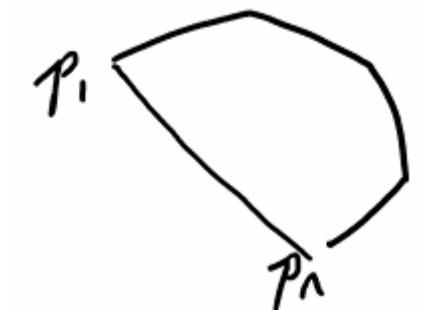
Ch 08 几何算法

基本定义

点p由坐标(x,y)表示。**直线**由点p和q所组成的点对表示，记作-p-q-。
线段由端点p和q所组成的点对来表示，记作p-q。**路径**P由点的序列 p_1, p_2, \dots, p_n 和连接它们的线段 $p_1-p_2, p_2-p_3, \dots, p_{n-1}-p_n$ 组成，有时将路径中的线段称为**边**。**封闭路径**是指终点和起点相同的路径，封闭路径也可称为**多边形**，定义多边形的点被称为**多边形的顶点**。多边形由顶点的序列来表示，改变它们的顺序会生成不同的多边形。**简单多边形**是指对应的路径不自交的多边形，即除了相邻边在公共顶点相交以外，其余边均不相交。简单多边形围成的区域一般称为**多边形的内部**。**凸多边形**是指满足以下条件的多边形，即连接多边形内部两点的任何线段，其本身全部位于多边形的内部。**凸路径**是指由点 p_1, p_2, \dots, p_n 所组成的路径，使得连接 p_1 和 p_n 能生成一个凸多边形。



简单多边形 非简单多边形 非凸多边形



凸路径

判定点是否在多边形内部

问题：给定简单多边形P和点q，判定该点是在多边形内部，还是在多边形外部。

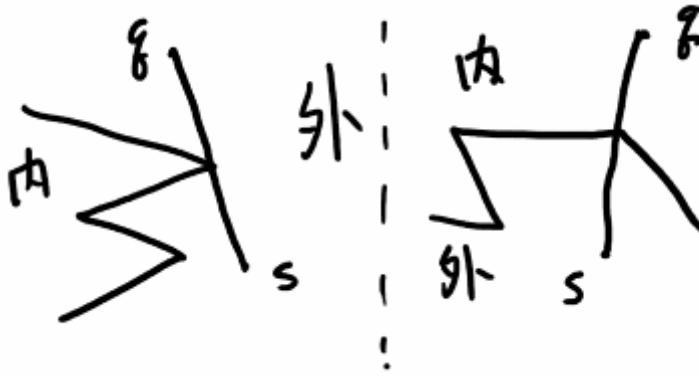
一般说来，点在多边形的内部当且仅当由多边形外部任意一点到此点的连线与多边形的交点个数是奇数。

```

Algorithm Point_in_Polygon_1(P,q)
Input: P (a simple polygon with vertices p1,p2,...,pn and edges e1,e2,...,en), q
(a point)
Output: Inside (a Boolean variable that is set to true if q is inside P and
false otherwise)
begin
    Pick an arbitrary point s outside the polygon;
    Let L be the line segment q-s;
    count:=0;
    for all edges ei of the polygon do
        if ei intersects L then increment count;
        if count is odd then Inside:=true
        else Inside:=false;
    end

```

存在特例，通过顶点时算几次？



如何选择一个在多边形外的点？

```

Algorithm Point_in_Polygon_2(P,q)
Input: P (a simple polygon with vertices p1,p2,...,pn and edges e1,e2,...,en), q
=(x0,y0)(a point)
Output: Inside (a Boolean variable that is set to true if q is inside P and
false otherwise)
begin
    count:=0;
    for all edges ei of the polygon do
        if the line x=x0 intersects ei then
            Let yi be the y coordinates of the intersection between the line
            x=x0 and ei;
            if yi<y0 then increment count
        if count is odd then Inside:=true
        else Inside:=false;
    end

```

算法复杂度：计算平面中两线段的交点需要花费常数时间，该算法计算n次这样的交点（此处n指多边形的大小），而执行其他操作需要花费常数时间。因此，该算法总共运行时间是O(n)。

构造简单多边形

问题：给定平面中n个点，将它们连接起来形成一条简单的封闭路径。

考察包含所有点的大圆C，通过始于C中心的旋转线对C的区域进行扫描，暂时假定旋转线每一时刻至多扫描经过一个点。如果按照扫描经过的先后顺序将这些点连接起来，就会得到一个简单多边形。

存在特例!

选取点集中x坐标最大的点z (如果多个点有最大x坐标, 选取其中y坐标最小的点, 极端点) 作为圆心。

仍有特例!

```
Algorithm simple_polygon(p1,p2,...,pn)
Input: p1, p2, ..., pn (points in the plane)
Output: P (a simple polygon whose vertices are p1, p2, ..., pn in some order)
begin
    choose the point with the largest x coordinate (and smallest y coordinate if
    there are several points with the same largest x coordinate) as p1
    for i:=2 to n do
        compute the angle ai between the line -p1-pi- and the x axis;
        Sort the points according to the angle a2,...,an;
        P is the polygon defined by the list of the points in sorted order
    end
```

如何计算角度? 角度相同时如何计算距离?

优化改为计算斜率, 计算平方和, 角度相同时可以按照由远到近排序

算法复杂度: 该算法的运行时间主要由排序所决定, 时间为 $O(n\log n)$.

凸包

点集的凸包为包围所有点的最小凸多边形。

问题: 计算平面中给定n个点的凸包。

凸包的顶点均来自集合中的点, 如果一点是凸包的顶点, 则称其属于凸包。

直接方法

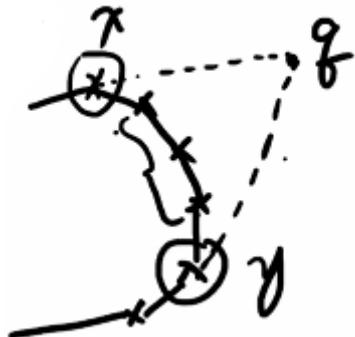
1. 得到3个点的凸包
2. 假定已经知道如何寻找 n 个点的凸包, 接着尝试寻找 $n+1$ 个点的凸包。
这第 $n+1$ 个点如何改变由先前 n 个点所形成的凸包呢?
 - 1) 点落在凸包内部, 凸包维持不变;
 - 2) 点落在凸包外部, 在这种情况下, 需要将凸包“延拓”以达到该点。

所以, 需要解决两个子问题: 判定一个点是否在凸包中, 以及当该点在凸包外部时能够延拓这个凸包, 这些问题并不容易。

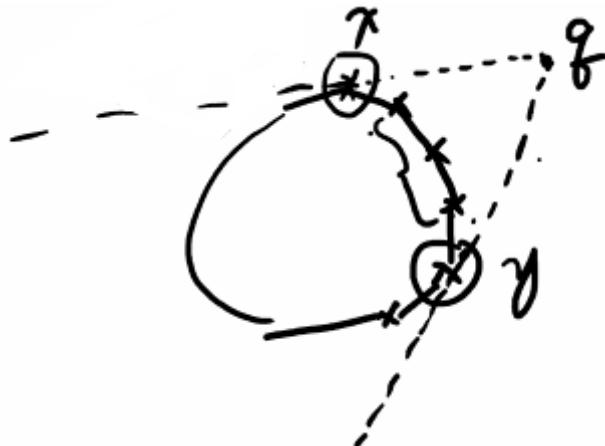
可能的改进: 1) 尝试选择在凸包中的点作为第 $n+1$ 个点; 2) 选择类似于最大点或者最小点的极端点 (extreme point)

再一次选取具有最大x坐标值的点 (如果多个点有着相同的最大的x坐标值, 则选取其中y坐标最小的点), 将这个点标记为 q , q 必为凸包的顶点。现在考虑如何修改(延拓)凸包使之包括 q 。

首先, 寻找原先属于旧凸包并且现在位于新凸包内部的点并将它们删除; 接着, 在两个现有顶点之间将这个新点作为新顶点插入。



即寻找点x和y，线xy一侧顶点全部删除，另一侧顶点全部保留



凸多边形的支撑线 (supporting line) 是指和多边形恰好只有一个交点的直线，即多边形全部位于支撑线的一侧。qx和qy即是支撑线

通常情况下，多边形中只有两个顶点和q存在连线，这些连线是支撑线，多边形位于两条支撑线之间，而这恰好就是想要的修改方式。在所有多边形中的点和q的连线中，两条支撑线和x轴之间分别有**最大**和**最小**角度。为了寻找这两个顶点，需要考虑从q到所有顶点的连线，计算它们的角度，并选取最大值和最小值。一旦识别了这两个极端顶点，就可以构造修改后的凸包了。

```

convex(n)
{
    if n=3 return 三角形
    找出x坐标最大的点，设为pn
    P=convex(n-1)为前n-1个点构成的凸包
    求pn到P所有顶点的角度，求出最大和最小值
    修正P的顶点列表得到新凸包P'
    return P'
}

```

算法复杂度：对于每个点，需要计算其和先前诸点的角度，寻找其中最大和最小角度，并从顶点列表中添加和删除点，所以处理第k个点的时间为O(k)。由递推关系 $T(n)=T(n-1)+O(n)$ 得到该算法的运行时间是 $O(n^2)$ 。

Gift Wrapping 礼品包裹算法

问题：如何避免构造一些含有不在最终凸包上的点的凸多边形？

若给定平面中n个点的集合，可以找到长度 $k < n$ 的凸路径，使它成为该点集凸包的一部分。

重点从延拓凸包变为延伸凸路径，即不再寻找更少点集合的凸包，而是寻找最终凸包的一部分。

从一个极端点（它必须在这个凸包上）出发，通过寻找支撑线的方法发现它在凸包上的相邻顶点，然后以相同方式从这些相邻顶点出发继续该过程。类似于拉隔离带的过程

```
Algorithm Gift_Wrapping(p1,p2,...,pn)
Input: p1,p2,...,pn (a set of points in the plane)
Output: P (the convex hull of p1,p2,...,pn)
begin
    set P to be the empty set;
    Let p be the point in the set with the largest x coordinate (and the
    smallest y coordinate if there are several points with the same
    largest x coordinate);
    Add p to P;
    Let L be the line containing p which is parallel to the x axis;
    while P is not complete do
        let q be the point such that the angle between the line -p-q- and L (in
        counterclockwise fashion) is minimal among all points;
        add q to P;
        L:=line-p-q-;
        p:=q
    end
```

算法复杂度：为了将第k个点添加进凸包，必须在n-k条直线中寻找最小（最大）角度。所以算法的运行时间是O(n²)，这并不比先前的延拓算法优越。

Graham扫描算法

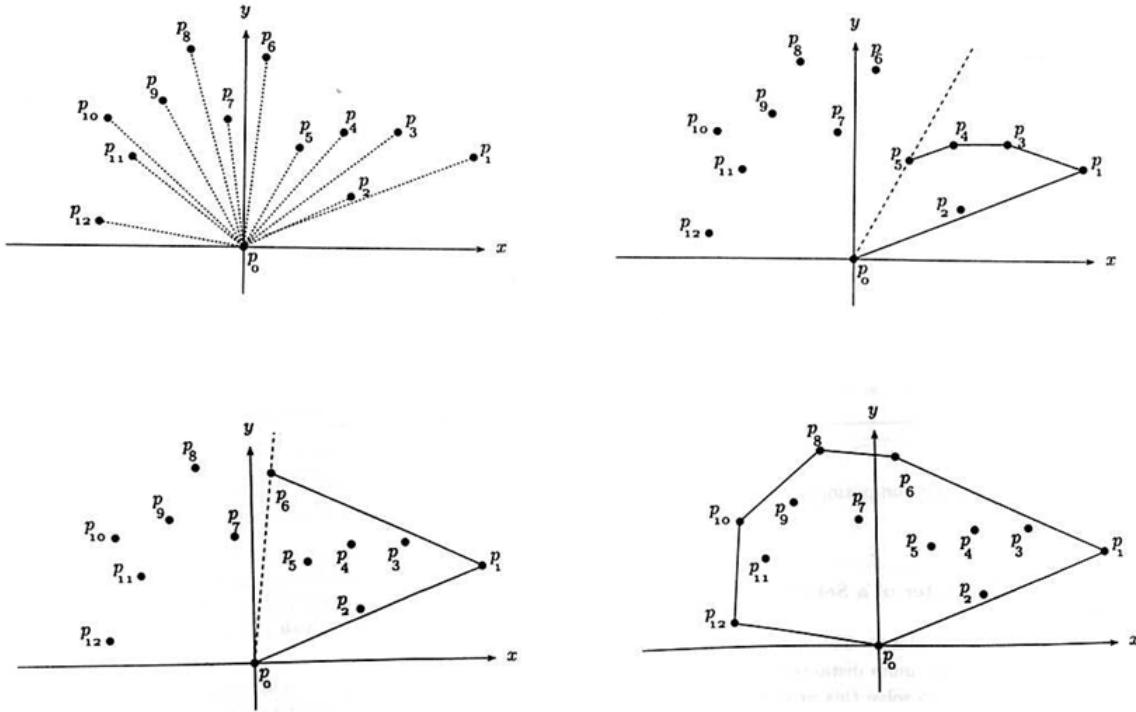
设p₁是有最大x坐标的点（如果多个顶点都有着相同最大x坐标值，则选取其中y坐标最小的点），对于每个点p_i，计算直线-p₁-p_i-和x轴之间的角度并根据这些角度将这些点排序。现在按照它们在多边形中的出现顺序扫描这些点，并且和先前一样尝试发现凸包的顶点。

如同Gift Wrapping，我们将构造一条由目前为止扫描过的点的一个子集所组成的路径，该路径是一条凸路径，其对应的凸多边形包围了所有目前为止扫描过的点，当所有点均被扫描到时就找到了这个凸包。

和Gift Wrapping的主要区别是，所维持的路径不一定是最终凸包的一部分，它只是目前为止所扫描到点的凸包的一部分，可能包含一些不在最终凸包中的点，但这些点将在以后除去。

归纳假设：给定平面中的n个点，并按照Simple_Polygon算法排序，我们可以在前k个点中找到一条凸路径，它所对应的凸多边形包围了前k个点。

用P=q₁,q₂,...,q_m表示从前k个点中所得到的凸路径，现在推广到k+1个点。考察直线-q_{m-1}-q_m-和-q_m-p_{k+1}-之间的角度，如果小于180度，那么p_{k+1}可以加入到现有的路径中。否则，q_m会位于P中除去q_m并加入p_{k+1}然后连接P₁和P_{k+1}后所得到的多边形的内部，但是修改后的路径不一定是凸的。必须继续检查路径的最后两条边，直至找到两条所成角度小于180度的边，这时路径为凸。



Algorithm Graham's_Scan(p_1, p_2, \dots, p_n)

Input: p_1, p_2, \dots, p_n (a set of points in the plane)

Output: q_1, q_2, \dots, q_m (the convex hull of p_1, p_2, \dots, p_n)

begin

 Let p_1 be the point in the set with the largest x coordinate (and the smallest y coordinate if there are several points with the same largest x coordinate);

 Use algorithm simple_Polygon to arrange the points around p_1 in sorted order; let the order be p_1, p_2, \dots, p_n ;

$q_1 := p_1, q_2 := p_2, q_3 := p_3$;

$m := 3$;

 for $k := 4$ to n do

 while the angle between $-q_{m-1}-q_m-$ and $-q_m-p_k-$ is ≥ 180 degrees do

$m := m - 1$;

$m := m + 1$;

$q_m := p_k$;

 end

算法复杂度：该算法的复杂度主要由排序所决定，所有其他步骤仅需要 $O(n)$ 时间，总共运行时间为 $O(n\log n)$ 。

最近点对

问题：给定平面中 n 个点，寻找相距最近的点对。

直接方法：检查所有的点对，然后取最小值。该方法需要 $n(n-1)/2$ 次距离计算以及 $n(n-1)/2-1$ 次比较。

使用归纳法的直接解法：先去除一个点，然后解决 $n-1$ 个点的问题，最后再考虑这个额外点。但是，若从 $n-1$ 个点的情况中所获得的唯一信息是最小距离，那么需要检查该额外点到其他所有 $n-1$ 个点的距离。因此，距离的计算量 $T(n)$ 满足 $T(n)=T(n-1)+n-1$ ，即 $T(n)=O(n^2)$ 。

最近点对的分治算法

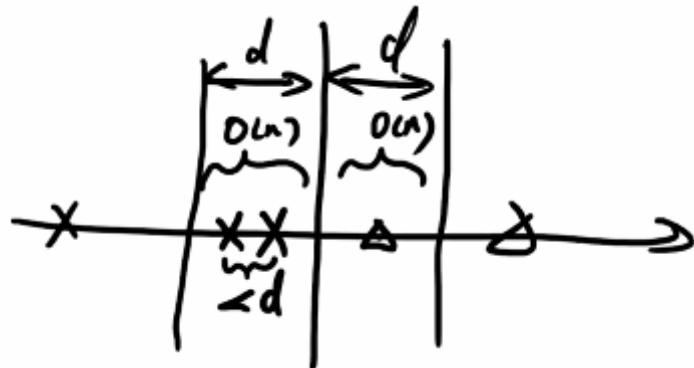
如果仅有两个点，那么可直接取它们之间的距离。

设P是n个点的集合并假定n是2的幂，首先将P划分成大小相等的两个子集P1和P2。根据归纳，寻找每个子集中的最近距离，假设P1和P2中的最小距离分别是d1和d2，并设d1≤d2。寻找整个集合的最短距离意味着必须检查是否存在P1中的点和P2中的点，满足两者之间的距离< d1。

最坏情况

$$\underline{T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)}$$

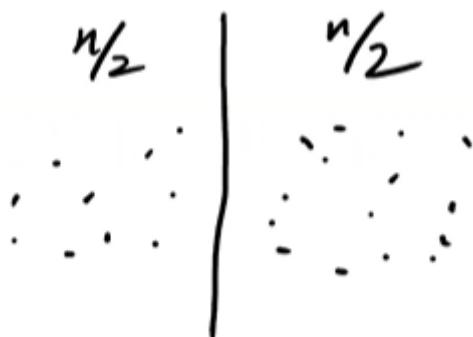
1D的情况



两侧的d区内都只有两个点

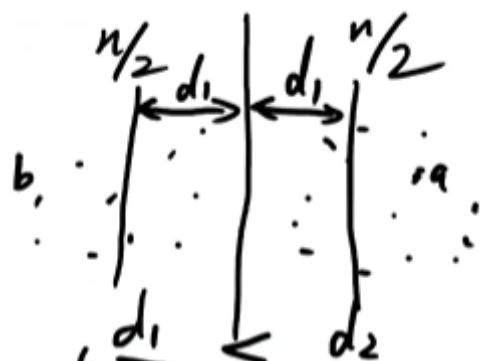
$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\ &= O(n \log n) \end{aligned}$$

2D的情况



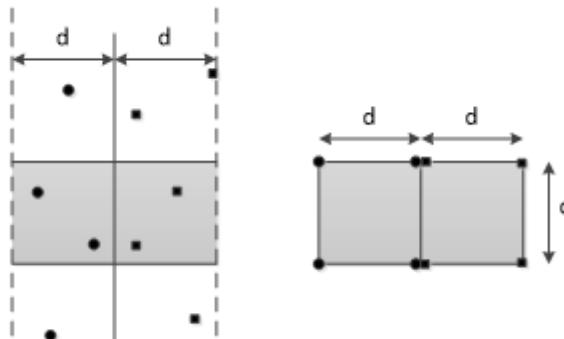
进行划分，如果进行排序，需要 $O(n \log n)$

选择快排找中位数， $O(n)$



首先，只需要考虑那些位于以两子集竖直分界线为中轴、宽度为 $2d_1$ 的带状区域（由两条宽度为 d_1 的带组成）中的点，在那个区域以外不可能存在点和另一个子集中的点距离小于 d_1 ，但是在最坏情况下，所有的点可能全部位于该带状区域。

观察：对于带状区域中的任何一点 p ，在另外一边只有很少数目的点和 p 之间的距离小于 d_1 ，这是因为在每条带中，所有的点至少相距 d_1 。如果带中点 p 的y坐标为 y_p ，那么在另外一边只需要考虑y坐标 y_q 满足 $|y_p - y_q| < d_1$ 的点，而在另一边带中最多只可能有6个这样的点。因此，若将带状区域中的所有点根据其y坐标排序并依次扫描这些点，那么对于每个点只需按照顺序检查其常数多个相邻点而非所有 $n-1$ 个点。



```

Algorithm Closest_Pair_1(p1,p2,...,pn)
Input: p1,p2,...,pn (a set of n points in the plane)
Output: d (the distance between the two closest points in the set)
begin
    Sort the points according to their x coordinates;
    Divide the set into two equal-sized parts;
    Recursively, compute the minimal distance in each part;
    Let d be the minimal of the two minimal distances;
    Eliminate points that lie farther than d apart from the separation line;
    Sort the remaining points according to their y coordinates;
    Scan the remaining points in the y order and compute the distances of each
    point to its five neighbors;
    if any of these distances is less than d then
        update d
end

```

算法复杂度：根据x坐标排序要花费 $O(n \log n)$ 步，但只需进行一次。解决两个规模为 $n/2$ 的子问题后，消除带状区域外的点可以在 $O(n)$ 步内完成，根据y坐标排序需要花费 $O(n \log n)$ 步。最后，扫描带状区域中的每个点，并将它依次和常数多个相邻点比较，需要花费 $O(n)$ 步。因此有 $T(n)=2T(n/2)+O(n \log n)$ ， $T(2)=1$ ，该递归关系的解为 $T(n)=O(n \log^2 n)$ 。

O(nlogn)的算法

原因：在组合步骤中花费O(nlogn)时间对y坐标排序

归纳假设：给定平面中 n 个点的集合，知道怎样找到最近距离以及怎样输出根据y坐标排序后的点集合。

不必在每一次组合解时都重新排序，只需要归并即可。递归关系变成 $T(n)=2T(n/2)+O(n)$, $T(2)=1$ ，从而可得 $T(n)=O(nlogn)$ 。

```
Algorithm Closest_Pair_2(p1,p2,...,pn)
Input: p1,p2,...,pn (a set of n points in the plane)
Output: d (the distance between the two closest points in the set)
begin
    Sort the points according to their x coordinates;
    Divide the set into two equal-sized parts;
    Recursively do the following:
        compute the minimal distance in each part;
        sort the points in each part according to their y ordinates;
    Merge the two sorted lists into one sorted list;
    {Notice that we must merge before we eliminate, we need to supply the whole
    set sorted to the next level of the recursion}
    Let d be the minimal of the two minimal distances;
    Eliminate points that lie further than d apart from the separation line;
    Scan the remaining points in the y order and compute the distances of each
    point to its five neighbors;
    if any of these distances is less than d then
        update d
end
```

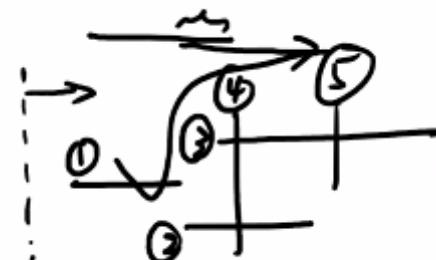
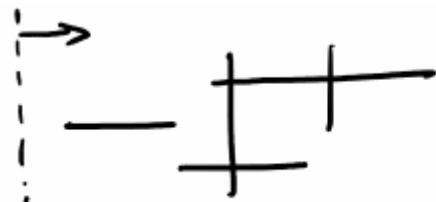
水平线段和垂直线段的交点

问题：给定平面中 n 条水平线段和 m 条垂直线段，寻找它们之间所有的交点。

为简单起见，假定任何两条竖直线段之间和任何两条水平线段之间均没有交点。

直接方法：每次求一条线段和所有其他线段进行比较，这样共需 $O(mn)$ 次比较。

扫描线技术：归纳的顺序取决于一条从左向右“横扫”平面的假想线（是无限直线，而非线段），线段按照其和这条假想线的相交顺序处理。



对所有线段的端点根据其x坐标进行排序。垂直线的两个端点由于有相同的x坐标，只需取其一即可，但是每条水平线的两个端点必须都用上。

由扫描顺序，出现在当前扫描线位置左方的所有交点均已知。

问题：检测交点应当发生在遇到垂直线时，还是在遇到水平线时？

当遇到垂直线时，与它相交的水平线仍然在考虑范畴之内（由于还没有到达其右端点）；而另一方面，当遇到水平线的左端点或者右端点时，或者还未曾遇到与之相交的垂直线，或者已经将那些垂直线忘了。

假定当前扫描线位于垂直线L的x坐标处，寻找到所有与L相关的交点需要哪些信息？

由于当前扫描线左方的所有交点假定都已知，不需要进一步考虑那些右端点位于扫描线左方的水平线。因此，只要考虑那些左端点位于扫描线左方，而右端点位于扫描线右方的水平线，需要维持这些水平线的列表。当遇到L时，需要检查它和这些水平线是否有交点，这里很重要的一点是，不需要通过检查x坐标来判断这些水平线是否和L相交！由于已经知道所有列表中的水平线的x坐标均与L相符，只要检查这些列表中的水平线的y坐标是否符合L的y坐标。

归纳假设：给定上述k个x坐标的列表（ x_k 为最右的坐标），知道如何得到 x_k 左方所有线段间的交点，并去除那些位于 x_k 左方的水平线。

称在考虑范畴内的水平线为候选线，它们是那些左端点位于当前x坐标左方，而右端点位于当前点x或其右方的水平线。

1. 第 $k+1$ 个端点是一条水平线的右端点：将这条线从候选线的集合中删除即可。
2. 第 $k+1$ 个端点是一条水平线的左端点：将这条线添加入候选线的集合。
3. 第 $k+1$ 个端点是一条垂直线：通过检查候选线集合中所有水平线的y坐标和该垂直线的y坐标，可以找到所有涉及该垂直线的交点。

最坏情况下仍然需要 $O(mn)$ 次比较操作。

将垂直线的y坐标和那些候选线集合中水平线的y坐标之间的比较操作数目最小化。

设当前所考虑的垂直线的y坐标为 y_L 和 y_R ，候选线集合中水平线的y坐标为 y_1, y_2, \dots, y_k ，而候选线集合中水平线根据其y坐标排序（即 y_1, y_2, \dots, y_k 依次递增）。通过执行两次二分搜索（对 y_L 和 y_R 各一次），可以找到所有与该垂直线相交的水平线。假设 $y_i < y_L \leq y_{i+1} \leq y_j \leq y_R < y_{j+1}$ ，与该垂直线相交的水平线恰好就是 $y_{i+1}, y_{i+2}, \dots, y_j$ 。也可以只对 y_L 进行一次二分搜索，然后顺序扫描y坐标直至找到 y_j 。

需要的数据结构：能够允许高效地插入新元素，删除元素，以及执行一维区间查询，如平衡树。

```
Algorithm Intersection((v1, v2, ..., vm), (h1, h2, ..., hn))
Input: v1, v2, ..., vm (a set of vertical line segments), h1, h2, ..., hn (a set of
horizontal line segments)
Output: The set of all pairs of intersecting segments {yB(vi)/yT(vi)} denote the
bottom/top coordinates of line vi)
begin
    sort all x coordinates in increasing order and place them in Q;
```

```

V:=空集; {V is the set of horizontal lines that are currently candidates for
intersection; it is organized as a balanced tree according to the y coordinates
of the horizontal lines}
while Q is not empty do
    remove the first endpoint p from Q;
    if p is the right endpoint of hk then
        remove hk from V
    else if p is the left endpoint of hk then
        insert hk into V
    else if p is the x coordinate of a vertical line vi then
        perform a one-dimensional range query for the range yB(vi) to yT(vi)
in V
end

```

算法复杂度：根据x坐标排序需要 $O((m+n)\log(m+n))$ 。每次插入和删除操作需要 $O(\log n)$ ，所以处理水平线总共需要的运行时间为 $O(n \log n)$ 。处理垂直线需要一维区间查询，它可以在 $O(\log n + r)$ 内完成，其中 r 是与这条直线相关的交点个数。该算法的总运行时间为 $O((m+n)\log(m+n)+R)$ ，其中 R 是交点的总数。

扫描线算法

两个基本点：

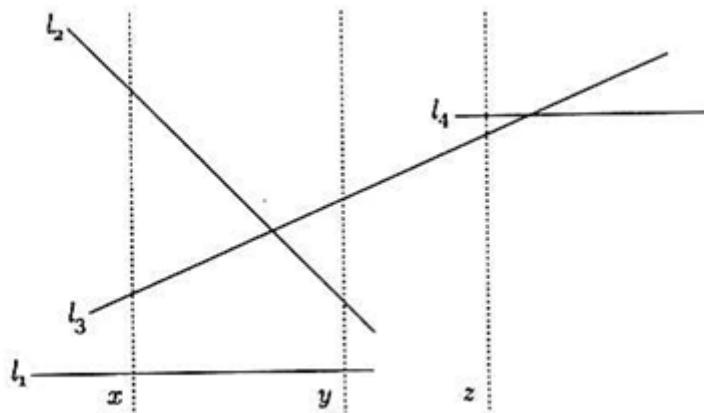
- 1) 事件点进度表：存储点的序列，该序列规定了扫描线停留的位置。该进度表可能会动态更新
- 2) 扫描线状态：是对扫描线上几何对象特性的描述

归纳的次序

线段交点

给定平面上n条线段构成的集合 $L=\{l_1, \dots, l_n\}$ ，求这些线段的交点。在这些线段中，不存在垂直线段，没有3条线段交于一点

above: $>_x$



扫描线算法

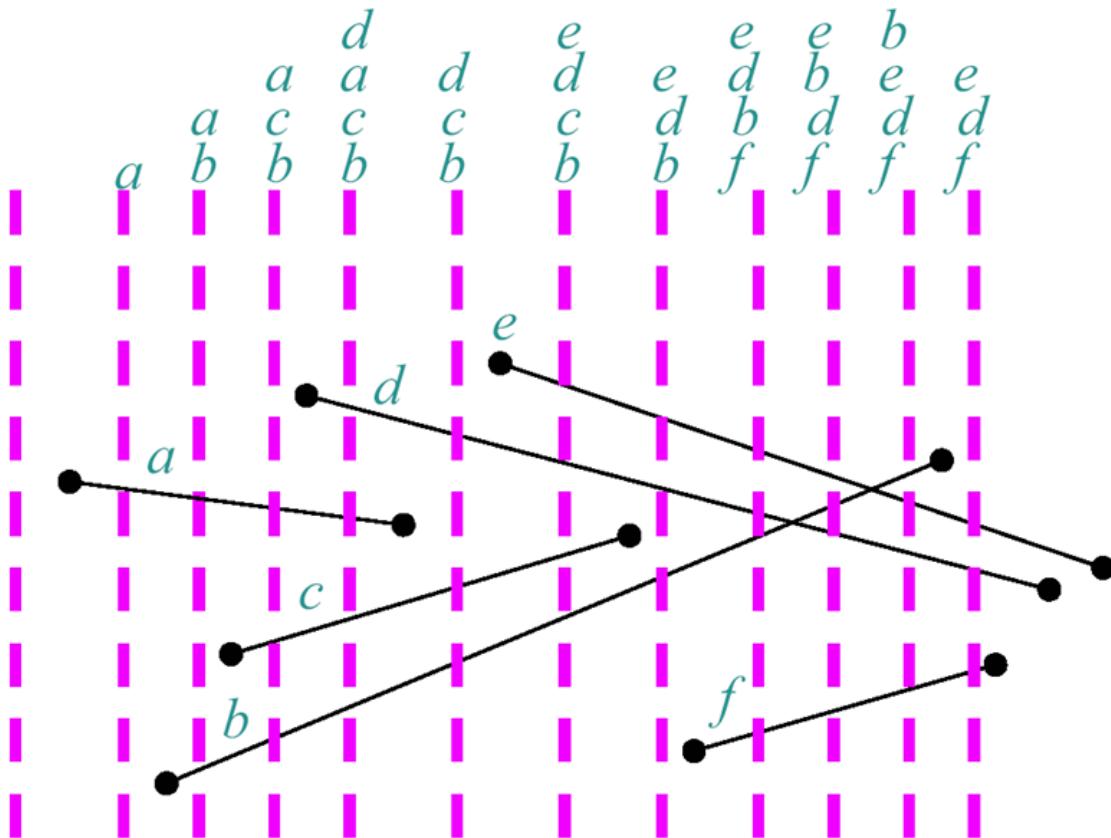
一条垂直扫描线从左往右扫描（即随时间改变x坐标）

扫描线状态集合S：是一个动态集合，描述扫描线状态，包含当前与扫描线交叉的线段，这些线段按照交点的y坐标排序。该顺序是动态变化的，会在以下情况发生改变：

遇到新的线段

现有线段结束

两条线段交点



事件点：线段端点及线段交点，并按点的x坐标排序，依次在这些点进行处理：

对于线段s的左端点：

将线段s加入动态集合S中

检查s在S中的相邻线段与s是否有交点

对于线段s的右端点：

从动态集合S中删除线段

检查s在S中的相邻线段之间是否有交点

对于线段s和t的交点：

交换s和t的次序

检查s, t和它们的相邻线段是否有交点

Input: A set $L = \{l_1, \dots, l_n\}$ of n line segments in the plane

Output: The intersection points of the line segments in L

Sort the endpoints in nondecreasing order of their x-coordinates and insert them into a heap E (the event point schedule)

while E is not empty

$p \leftarrow \text{delete-min}(E)$

if p is a left endpoint then

let l be the line segment whose left endpoint is p

$\text{insert}(l, S)$, $l_1 \leftarrow \text{above}(l, S)$, $l_2 \leftarrow \text{below}(l, S)$

if l intersects l_1 at point q_1 then process(q_1)

if l intersects l_2 at point q_2 then process(q_2)

else if p is a right endpoint then

```

let l be the line segment whose right endpoint is p
l1<-above(l,s), l2<-below(l,s), delete(l,s)
if l1 intersects l2 at point q to the right of p then process(q)
else {p is an intersection point}
    Let the two intersecting line segments at p be l1 and l2 where l1 is
    above l2 to the left of p
    l3<-above(l1,s), l4<-below(l2,s)
    if l2 intersects l3 at point q1 then process(q1)
    if l1 intersects l4 at point q2 then process(q2)
end if
end while

```

Time: $O((2n+m)\log(2n+m))$ m是交点个数, 循环 $2n+m$ 次, $2n+m$ 个点进堆

平面图, 多边形和美术馆问题

平面图

图由节点和边构成

平面图是指其可以在平面上绘制且边之间除了端点以外不相交

平面直线图(planar straight line graph , PSLG): 边皆为直线的平面图。

点vertex, 边edge, 面 face

总有一个面是无界的

Euler公式: $V - E + F = 2$

$V - E + F$ 称为Euler特征, 在平面上是常量。

若允许图不连续, 记C为连通分支数, 则有

$V - E + F - C = 1$



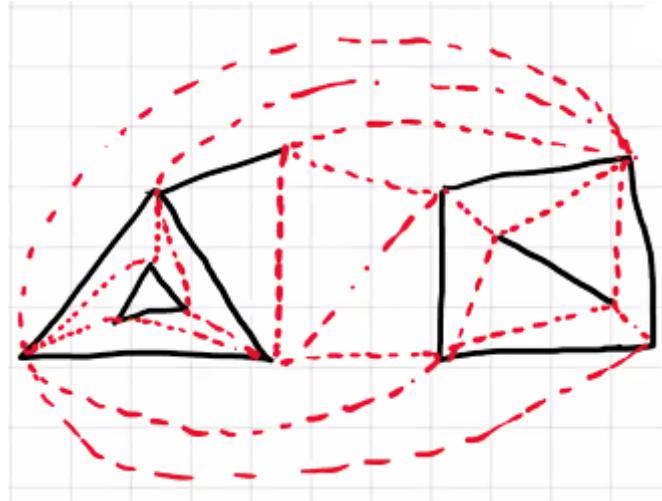
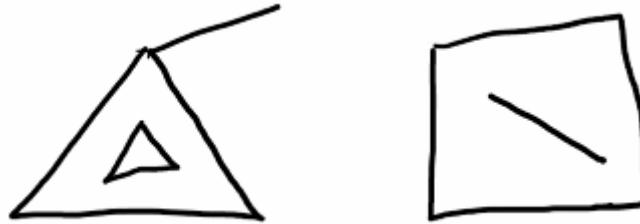
$$\begin{array}{ll} V=7 & E=7 \\ F=3 & C=2 \end{array}$$



$$\begin{array}{ll} V=6 & E=5 \\ F=2 & C=2 \end{array}$$

定理: 具有V个顶点的平面图具有最多 $3(V - 2)$ 条边和最多 $2(V - 2)$ 个面

证明: 三角划分, 对所有边数大于3的面插入新边, 使每个面 (包括最外边的面) 都是三条边



$$G \rightarrow G'(V, E')$$

G' : 只有一个连通分支. 每个面三条边
每条边两侧是两个不同的边

$$3F' = 2E' \quad *$$

$$\begin{cases} 3F' = 2E' \\ V - E' + F' = 2 \end{cases} \quad *$$

$$V - \frac{3}{2} F' + F' = 2 \quad F' \leq F = 2(V-2)$$

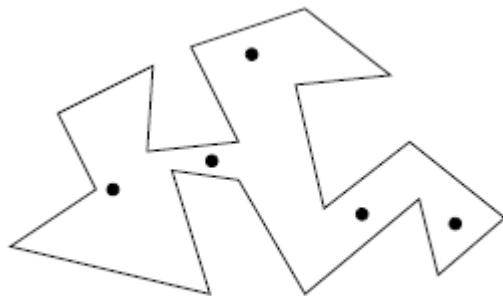
顶点数、边数和面数之间的线性关系只在2维平面中成立，在3维空间中n个顶点的图可能会有 $\Theta(n^2)$ 条边

美术馆

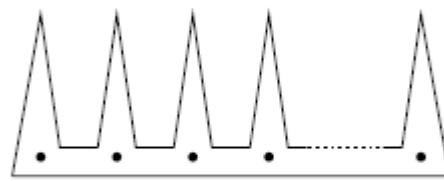
在简单多边形中的两个点x和y能够相互看见是指x和y之间的线段落在多边形的内部。该线段的两个端点可以落在多边形边界上，但该线段不能穿越任何顶点或边

问题：如果美术馆的平面图可以表示为n个顶点的多边形，那么至少需要多少名保安以监控该美术馆（即美术馆中任意一点都能够被保安看到）？

注意：问题中只告诉了多边形的顶点数，并未告知其结构。我们需要求出至少需要多少名保安，才能确保监控任意的n个顶点的多边形，而不是对于给定的多边形求出所需最少的保安数量



A guarding set



A polygon requiring $n/3$ guards.

定理（美术馆定理）：对于任意的有n个顶点的简单多边形，存在符合监控要求的保安集合，其中保安个数最多为 $\lfloor n/3 \rfloor$

简单多边形的三角划分是指对多边形内部的分割，分割后得到的每个区域都是三角形，这些三角形的顶点为原多边形的顶点。

对角线/弦：是多边形三角划分的重要概念，是连接多边形两个顶点的线段，该线段位于多边形内部，且相互之间不相交。

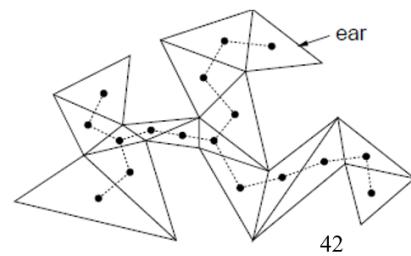
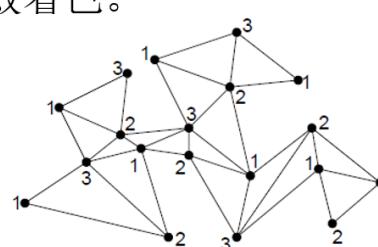
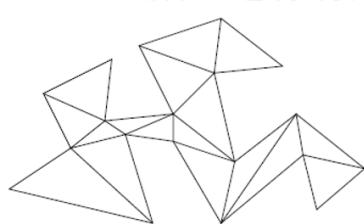
三角划分可以视为求多边形最大的不相交对角线集合

思考题：对凸多边形，1) 有多少种三角划分的方法？2) 如何使对角线之和最小？

引理： n个顶点的简单多边形存在三角划分，该划分具有 $n-3$ 条对角线， $n-2$ 个三角形

对于平面图G，存在平面图G*为G的偶图，G*的顶点代表 G的面，如果G的两个面有公共边，则G*对应的两个顶点之间存在边。

引理： T是简单多边形三角划分后得到的图，则T可用3种颜色进行有效着色。



引理1证明

$n=3$ 0条对角线，1个三角形

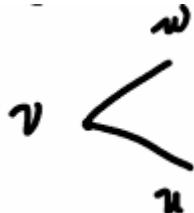
$n=4$ 1条对角线，2个三角形



1. 假设 $< n$ 时命题成立
2. 有 n 个顶点时，先加入一条对角线，使多边形一分为二。分出的两个多边形分别有 m_1 和 m_2 个点
 $m_1+m_2=n+2$ 且 $m_1, m_2 < n$
由归纳假设，这两个多边形各有 m_1-2 和 m_2-2 个三角形，各有 m_1-3 和 m_2-3 条对角线
三角形总数为 $m_1+m_2-4=n-2$ ，对角线总数为 $m_1+m_2-6+1=n-3$

补充证明有 n 个顶点时，先加入一条对角线，可以使多边形一分为二

假设 v 是最左侧顶点， u, w 是相邻顶点



若 $u-w$ 在多边形内部，则将多边形一分为二

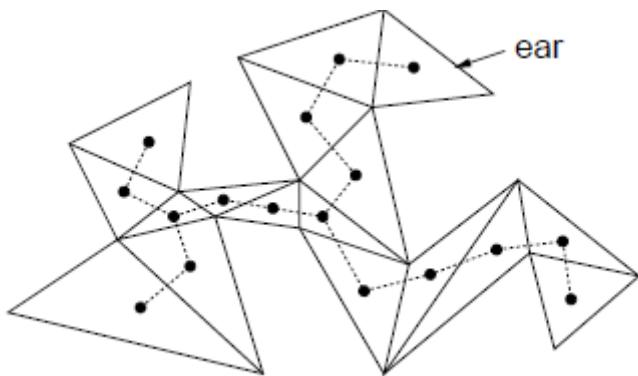
若 $u-w$ 在多边形外部



必然有多边形的顶点在三角形uvw内部，选择其中离u-w最远的一个点 v' ， $v-v'$ 与多边形不会有交点。

引理2证明

对于图T，构造偶图 T^*



T^* 是一棵树：由于每一条对角线都可以将多边形T一分为二，所以 T^* 中每一条边都是割边

T^* 最大度数为3，因为三角形最多只有三个边

对T的三角形个数做归纳

1. $T=$ 三角形，成立
2. 假设三角形个数 $< n$ 时成立

当T划分后有 n 个三角形时，对应的 T^* 至少有一片叶子，删去此叶子意味着T中去掉一个三角形ear且该三角形与T中其余部分共享一条边。

由归纳假设，T的其余部分可以三着色，放回刚才删除的三角形，其中两个顶点已着色，剩下顶点只和这两个顶点相连，可用第三种颜色着色

n 个顶点用了3种颜色，有一种颜色出现的次数是不超过 $n/3$ 的，将保安置于这些顶点上，就可以确保每个三角形都被覆盖，从而整个多边形被覆盖，证明了解的存在性。

多边形三角划分

简单多边形的三角划分有很多应用，在某些应用中对三角形的形状有要求，我们暂不考虑这些要求

前面证明美术馆定理时证明有对角线可将多边形一分为二，这可以推导出一种三角划分算法，时间复杂性 $O(n^2)$

多年来此问题有 $O(n \log n)$ 算法

1991年， $O(n)$ 算法，相当复杂，实用中反而慢

两步走

1) 单调多边形的三角划分， $O(n)$

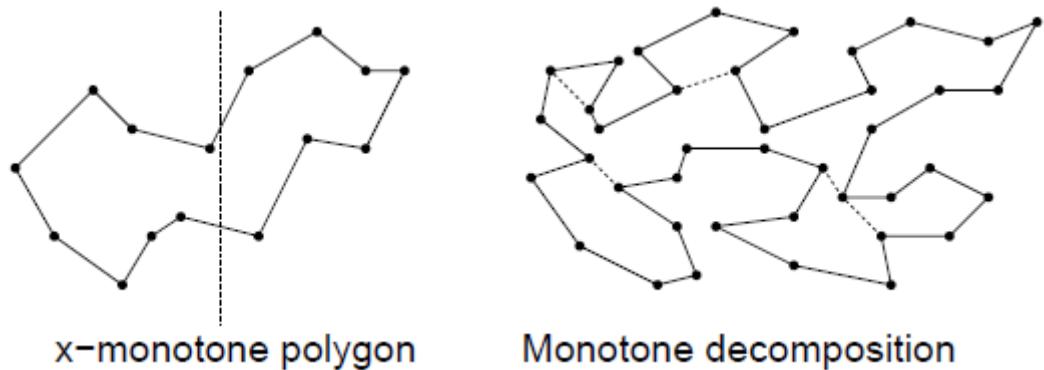
2) 任意多边形划分为单调多边形， $O(n \log n)$

单调多边形

折线 C 被称为关于给定直线 L 是严格单调的是指对于任意与 L 垂直的直线与 C 最多只有一个交点

折线 C 被称为关于给定直线 L 是单调的是指对于任意与 L 垂直的直线如果与 C 相交，则相交部分可以是单个点，也可以是一条边

多边形 P 关于直线 L 单调是指其边界 ∂P 可以被分为两条折线，每条折线关于 L 都是单调的



考虑关于x轴单调。称这种多边形为水平单调。

很容易测试多边形是否水平单调。方法如下：

(a) 在 $O(n)$ 时间内找到最左和最右顶点（ x 坐标最小和最大）

(b) 这两个顶点将多边形的边界分为两条折线：上折线和下折线。沿着每条折线从左向右遍历，检查其 x 坐标是否非递减。这一步耗时 $O(n)$

思考：如何验证一个多边形沿任一方向单调？算法复杂性 $O(n)$

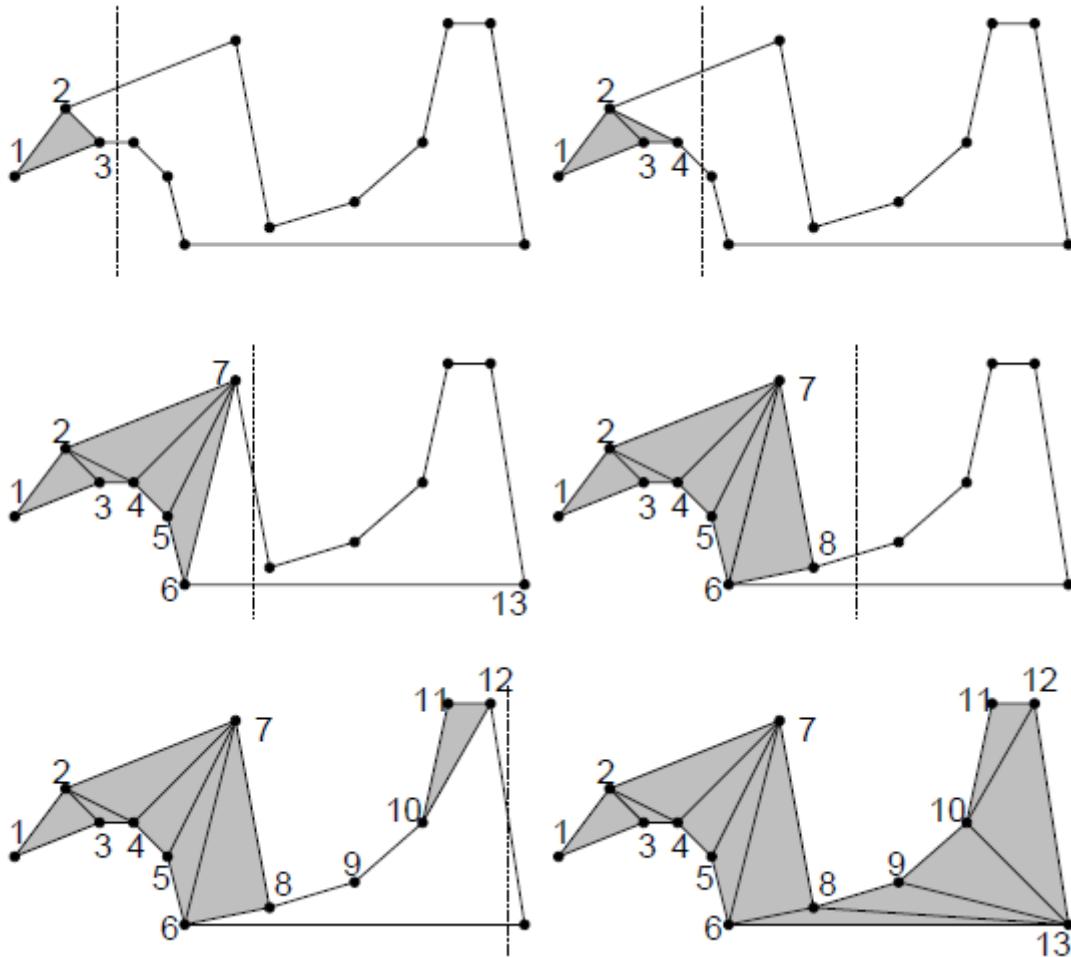
单调多边形的三角划分

对单调多边形可以使用扫描线算法进行三角划分

假设多边形的顶点按照其 x 坐标递增的顺序排序

注意：这一假设不需要通过排序完成。提取上折线和下折线，然后利用类似mergesort的方法对顶点进行合并，时间复杂性为 $O(n)$

三角划分的思想：通过增加对角线的方法将当前顶点左侧部分三角化，并将得到的三角区域丢弃，在后面的计算过程不再考虑



算法有效性的基础是每当遇到一个顶点，其左侧未三角化部分具有一种非常简单的结构。该结构使我们可以在常数时间内来决定是否增加对角线。通常我们可以在常数时间内增加对角线。由于对角线的数量为 $n-3$ ，因此算法的时间复杂性为 $O(n)$

引理：对于 $i \geq 2$ ，令 v_i 为算法刚处理过的顶点。在 v_i 左侧未三角化区域包含两条交于 u 的 x 轴单调折线，一条上折线，一条下折线，每条折线都至少有一条边。如果 v_i 到 u 的折线含有两条及以上的边，那么这条折线构成反射型折线，即相邻两条边构成的内部角至少是180度，此时另外一条折线只有一条边，该边的左端点为 u ，右端点位于 v_i 的右方

归纳假设：

1. $i=2$ $u=v_1$
2. 假设 v_{i-1} 时命题成立，考虑 v_i 。
 1. v_i 与 v_{i-1} 不在同一折线上设 u 为两条折线交点
 1. 若 u 和 v_{i-1} 间只有一条边，直线连接 v_i 和 v_{i-1} 并使 $u=v_{i-1}$



2. 否则加对角线连接 v_i 至反射型折线上所有点（除了 u ），这些点对 v_i 而言都是可见的，可以切出三角形。

由于该折线是反射型的，关于 x 轴单调且位于 v_i 左侧，所以该折线本身不会妨碍 v_i 连到次折线上的点，同时由于多边形单调，所以多边形未处理部分都在 v_i 的右侧，不会影响三角划分



完成上述工作后令 $u=v_{i-1}$ 命题成立

2. v_i 和 v_{i-1} 在同一条折线上



如果 $\alpha < 180^\circ$ 连对角线进行三角划分直至 u 到 v_i 是反射型折线或只有一条边



```

Assume that the vertices of the polygon have been sorted in increasing order of
their x-coordinates
Initialize an empty stack S, and push  $u_1$  and  $u_2$  onto it
for  $j=3$  to  $n-1$ 
    if  $u_j$  and the vertex on top of S are on different chains
        Pop all vertices from S
        Insert into D a diagonal from  $u_j$  to each popped vertex except the last
        one
        Push  $u_{j-1}$  and  $u_j$  onto S
    else Pop one vertex from S
        Pop the other vertices from S as long as the diagonals from  $u_j$  to them
        are inside P. Insert these diagonals into D. Push the last vertex that
        has been popped back onto S
        Push  $u_j$  onto S
Add diagonals from  $u_n$  to all stack vertices except the first and last one
  
```

时间复杂性 $O(n)$:

第1步：线性时间

第2步：常数时间

for循环：执行 $n-3$ 次。由于每次循环至多2个顶点进栈，因此加上第2步的进栈，总共进栈次数为 $2n-4$ ，由于出栈次数不能超过进栈次数，因此for循环总的运行时间为 $O(n)$

最后1步：线性时间

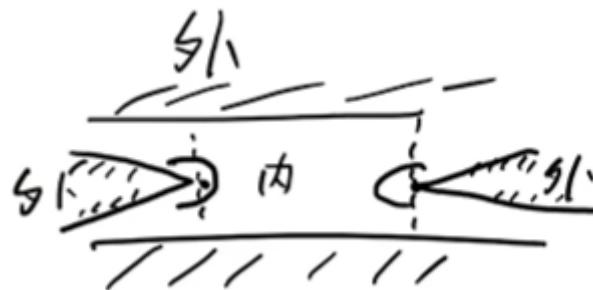
一般多边形的单调划分

为使用前述三角划分算法，首先需要将任意简单多边形划分为单调多边形。这一过程也可通过扫描线算法实现，算法将通过添加一系列互不相交的对角线将一般多边形划分为一些单调多边形

转向点：假设沿着上折线或下折线从最左面的顶点走到最右面的顶点，在此过程中如果在经过某个顶点时移动的方向由向左变成向右，或者由向右变成向左，则该顶点称为转向点

目标：通过添加对角线去除转向点

单调性在怎样的顶点不满足：内部角大于180度，并且两条边要么都在该顶点左侧（此时该顶点称为合并点），要么都在右侧（此时该顶点称为分离点）。



引理：多边形如果没有合并点和分离点，则该多边形是关于x轴单调的

证明非x轴单调的多边形，一定存在分离点或合并点

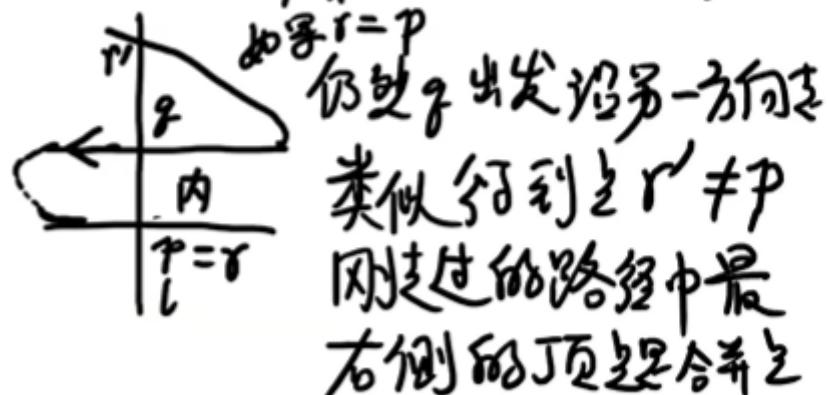
$\because P$ 关于 x 轴不单调

\therefore 存在垂直线 L 与 P 相交，得到多个线段
选择之，使最下处在多边形内的部分
是线段而非端点

设该线段
下方的端点为 p ，上方为 q


由 q 出发沿边走，且使 P
位于左侧，直至到达 p
 r 为止与 P 的交点
若 $r \neq p$

若 $r \neq p$, 则走过的路径
中最近左侧的顶点是分离点



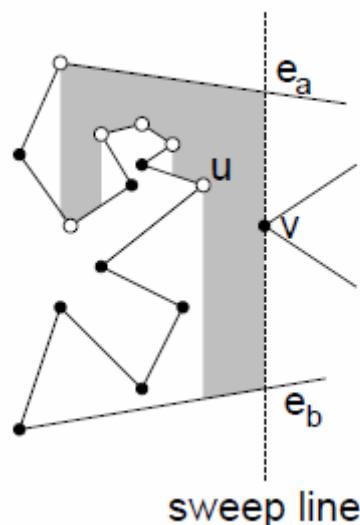
扫描时如果遇到分离点 v , 则存在边 ea 在其上方, 边 eb 在其下方

方法1: 将 v 与 ea 或 eb 的左端点相连。但有可能 v 看不到这两个顶点。



方法2: 找到 ea 和 eb 之间的顶点 u , u 和 v 相互之间能够看到

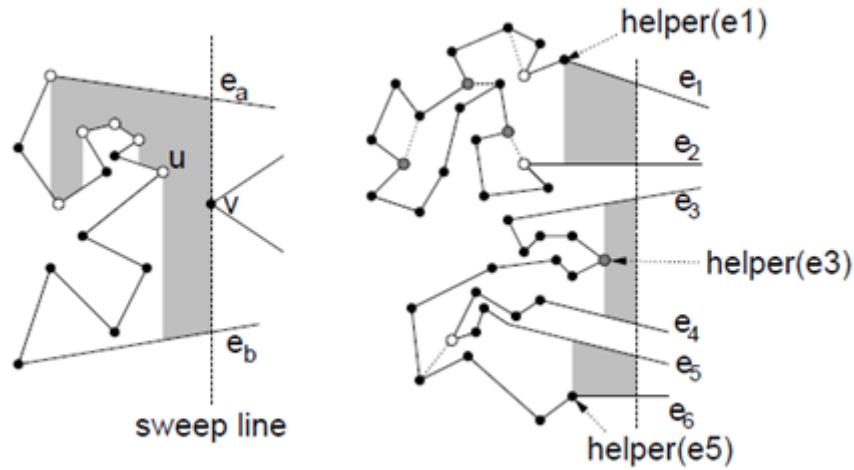
设想从 ea 有光线垂直照下, 所有能够被照到的顶点 (包括 ea 的左端点) 称为 ea 下垂直可见。在这些顶点中, 最右面的顶点 u 可以被 ea 与 eb 之间的扫描线上所有点看到



$helper(ea)$: 令 eb 为沿着扫描线在 ea 下方的第一条边, 则 $helper(ea)$ 是位于 ea 和 eb 之间的折线上能够被 ea 垂直可见的最右顶点

若 ea 是分离点上方第一条多边形的边, 连接分离点和 $helper(ea)$, 即可将该分离点去除

注: $helper(ea)$ 可能是 ea 的左端点。 $helper(ea)$ 的值依赖于扫描线的位置, 扫描线移动会改变 $helper(ea)$ 的值。另外 $helper$ 只对与扫描线相交的边有定义



事件点进度表

多边形的顶点

将这些点按x坐标递增的次序排序。在计算过程中无新事件，因此这些点可以存储在有序表中即可

扫描线状态

与扫描线相交的边的列表，顺序为自上而下

这些边存储为字典结构，从而可以在 $O(\log n)$ 时间内进行插入、删除、查找、前驱和后续等操作

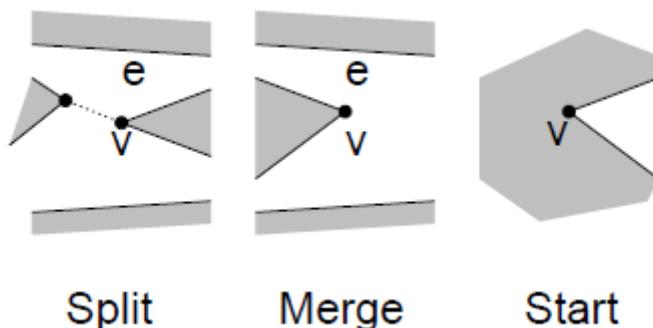
事件处理

根据顶点所属边的局部结构将所有点分为6类。假设当前扫描线遇到顶点v：

分离点：查找扫描线状态表，找出v上方第一条边e，增加对角线连接v和helper(e)，将以v为端点的两条边加入状态表，其中下面的一条边的helper定义为v，helper(e)也修改为v

合并点：在扫描线状态表中找出以v为端点的两条边并删除，令e为v上方第一条边，令helper(e)=v

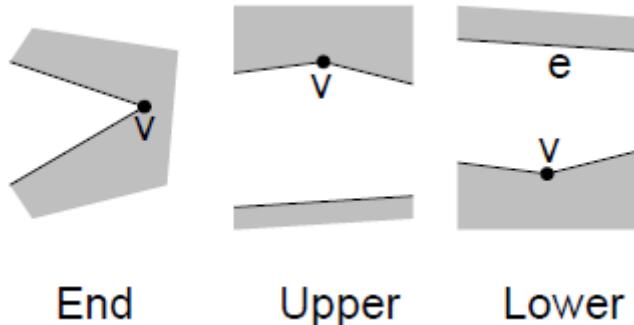
起点：两条边都在v右侧，且内部角小于180度。将两条边都加入状态表，其中上方的一条边的helper为v



终点：两条边都在v左侧，且内部角小于180度。从状态表中删除这两条边

上折线顶点：两条边在v左右两侧，其下方为多边形内部。在状态表中用右面的边替换左面的边。新加入的边的helper为v

下折线顶点：两条边在v左右两侧，其上方为多边形内部。在状态表中用右面的边替换左面的边。若e为其上方的边，则helper(e)=v



以上处理仅在分离点加入了对角线。那么如何消除合并点？

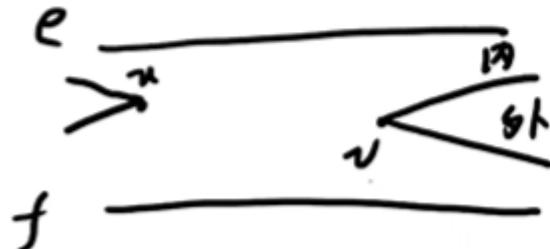
可以自右向左做类似处理，可以证明不会出现相互交叉的对角线，但有可能会得到两条相同的对角线（一条对角线恰好从左到右连接了合并点和分离点）

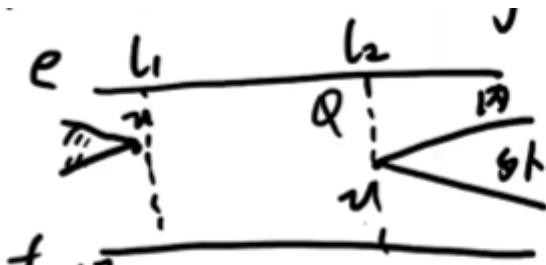
但是当改变helper(e)时，可以检查一下原来的值是否为合并点，如果是的话，加入新的对角线连接合并点和新的helper(e)。可以证明此方法与反向扫描等价，并且不会生成重复的对角线。

定理：上述算法通过增加一组互不相交的对角线，将简单多边形划分为一些单调多边形

证明：增加的线段不会与多边形的边相交，也不会与其他对角线相交

考虑对角线 u, v . 该线段
是扫描线到达 v 时添加的
设 e 是 v 上方线段. f 为 v 下方线段.





当扫描到达 v 时 $\text{helper}(e) = u$

① 证明 $u-v$ 不会与多边形边相交

考虑由边 u, v 的垂直线段 l_1, l_2

l_1, l_2, e, f 围成一个四边形 Q

则 Q 中不可能包含 P 的顶点.

否则 u 不能成为 helper

P 有边与 $u-v$ 相交?

由于 g 在 Q 内不能有顶点

且多边形边互不相交

$\therefore g$ 必定与 l_1 或 l_2 相交.

对于 u, v 而言, e 为其上方最近的边

\therefore 方便 \therefore 不可能有 P 的边与 $u-v$ 相交

② $u-v$ 不会与已添加的对角线相交.

由于 P 的顶点不在 Q 内, 且已添加

对角线顶点必在 v 的左方.

\therefore 已添加对角线不可能与 $u-v$ 相交

时间复杂性: 事件点为 n 个, 每个事件点处理时间为 $O(\log n)$, 因此将简单多边形划分为单调多边形需要 $O(n \log n)$, 将简单多边形三角化需要 $O(n \log n)$

相交半平面

半平面

平面上任意一条直线将平面一分为二，形成两个半平面，分别位于直线两侧。根据半平面是否包含该直线，可以将半平面分别称为闭的或开的

此处讨论闭半平面

表示

用以下方程表示平面上的直线

$$y = ax - b$$

其中 a 为斜率

更一般的直线表示需要3个参数：

$$ax + by = c$$

为表示闭半平面，将以上方程变换为不等式

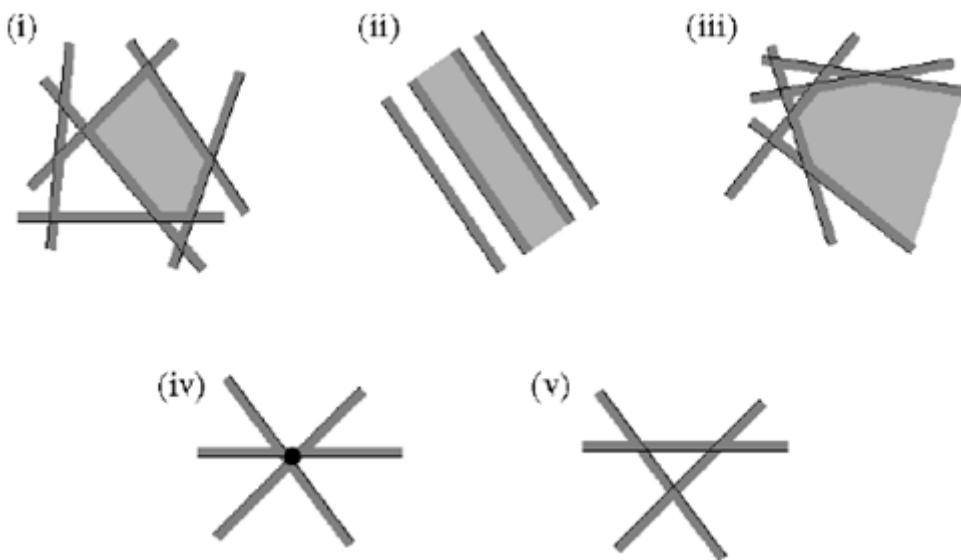
$$y \leq ax - b \text{ or } ax + by \leq c$$

其中第一个不等式表示直线下方的半平面，而后一个则更加一般化，可以表示任意一侧的半平面，只需将所有系数乘以-1即可

相交半平面问题

给定 n 个半平面 $H = \{h_1, h_2, \dots, h_n\}$ ，求交集

一个半平面（闭或开）是一个凸集，因此任意个半平面相交得到的仍然是一个凸集。与凸包问题不同， n 个半平面相交可能会得到空集或无界集合



n 个半平面的交集最多有多少条边？在最后的交集中，每个半平面最多只能贡献一条边，因此最多只能有 n 条边

计算半平面的交集能够多快？ $\Omega(n \log n)$.

分治法

- (1) If $n = 1$, then just return this halfplane as the answer.
- (2) Split the n halfplanes of H into subsets H_1 and H_2 of sizes $n/2$ 向下取整 and $n/2$ 向上取整, respectively.
- (3) Compute the intersection of H_1 and H_2 , each by calling this procedure recursively. Let C_1 and C_2 be the results.
- (4) Intersect the convex polygons C_1 and C_2 (which might be unbounded) into a single convex polygon C , and return C .

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

如果 $S(n)=O(n)$, 则总的计算时间为 $O(n \log n)$

是否可以用求线段交点的算法?

$$O((2n + m) \log(2n + m))$$

$$m=O(n)$$

新问题: 如何在 $O(n)$ 时间内计算两个凸多边形的交集

两个凸多边形的交集

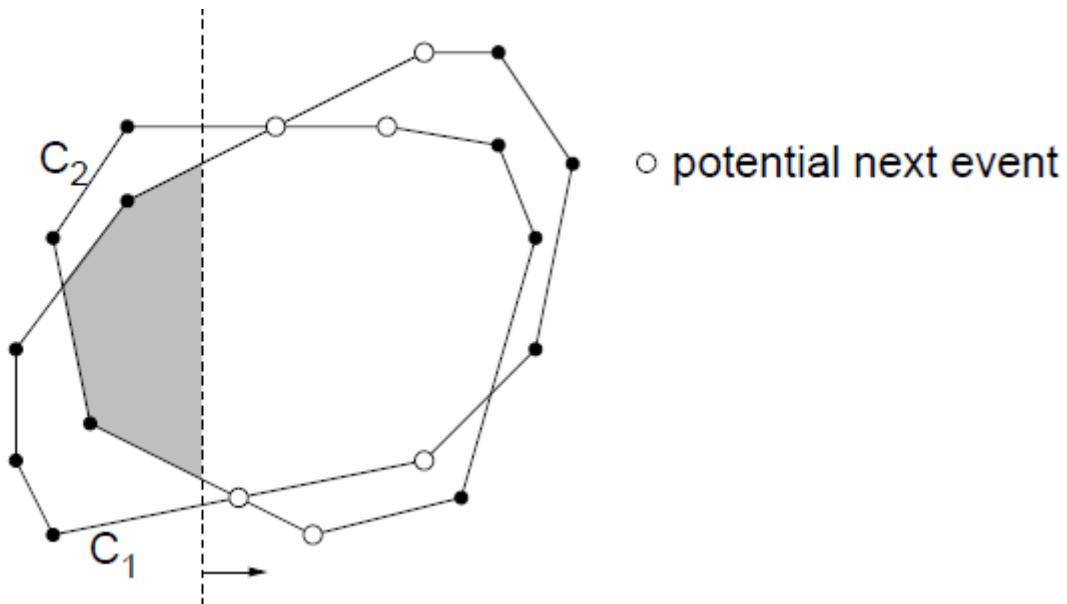
两个凸多边形的交集还是凸多边形

扫描线从左向右移动

由凸集的性质, 扫描线与一个凸多边形至多两个交点, 因此扫描线状态表中最多只需记住4条边

在任意一点, 下一个事件点最多只有8种可能性: 4条与扫描线相交的边的右端点, 这4条边的交点 (最多4个)。因此对其操作都可以在常数时间内完成。

因此求两个凸多边形的交集可以在 $O(n)$ 时间内完成



堆始终只有常数规模, 所以 $O(n)$

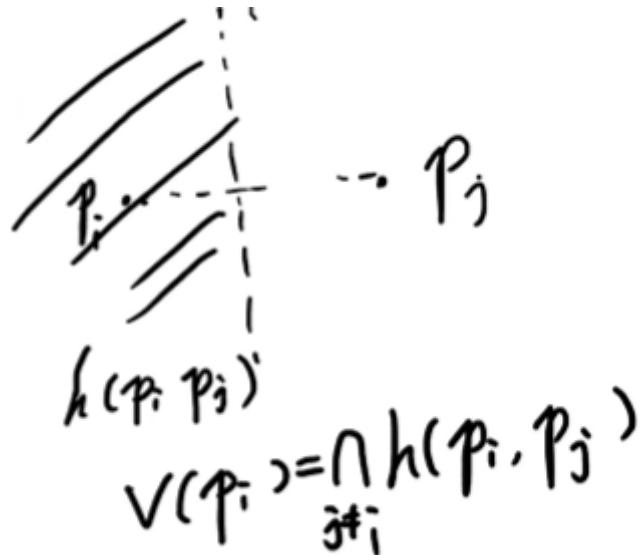
Voronoi图与Fortune算法

Voronoi图

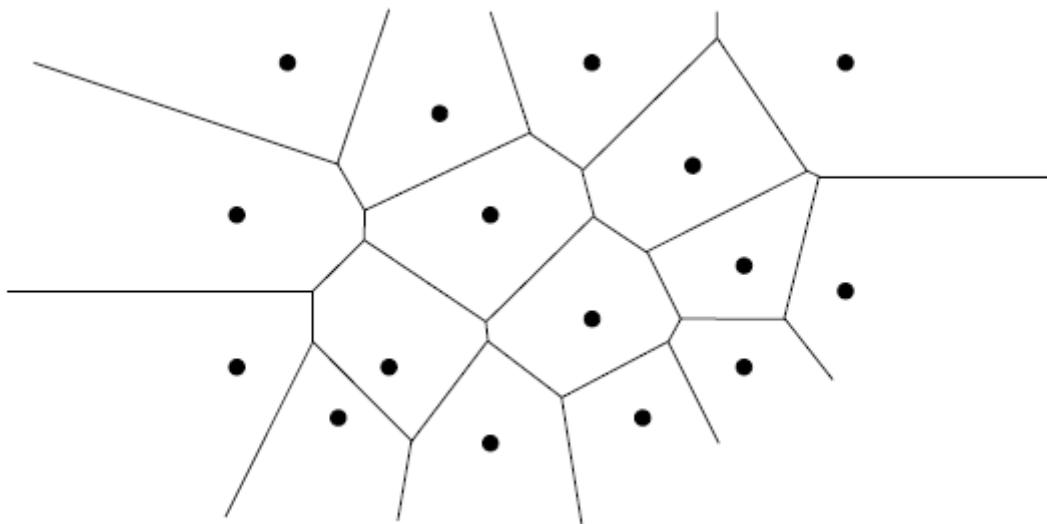
Voronoi图是计算几何学中一种很重要的结构，关注哪个点与哪个点更近

令 $P = \{p_1, p_2, \dots, p_n\}$ 为平面上一组点，这些点称为基点(site)。定义Voronoi区域(cell) $V(p_i)$ 为一个点集，其中的点与 p_i 之间的距离小于到其它基点的距离，即

$$V(p_i) = \{q \mid |p_i q| < |p_j q|, \forall j \neq i\}$$



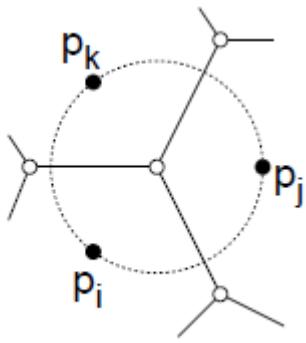
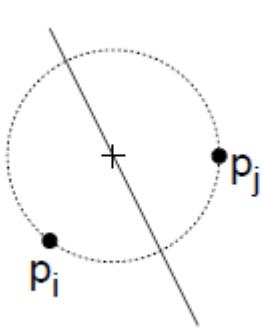
点集 P 的 Voronoi 图(diagram) $\text{Vor}(P)$ 被定义为平面上去除所有开Voronoi区域后剩余的部分。不难验证 Voronoi 图由一组线段(或射线和直线)构成



Voronoi边：边上的每个点距离其最近的两个基点 p_i 和 p_j 距离相同。如果以该点为圆心画一个圆，并且 p_i 和 p_j 在圆上，则圆的内部不会有其它基点

Voronoi顶点：位于三个Voronoi区域的交点，距离对应的三个基点的距离相等，如果这三个基点位于某个圆上，则Voronoi顶点为圆心，圆内部没有其它基点

度：如果没有4个基点在同一个圆上，则Voronoi图中的顶点的度数为3



凸包：Voronoi图中的区域是无界的当且仅当对应的基点在凸包上（一个基点在凸包上当且仅当它是距离无穷远处某个点最近的）。如果已知Voronoi图，则可以在线性时间内得到凸包

连通性： P 为平面上点集。若所有基点在同一条直线上，则 $\text{Vor}(P)$ 由 $n-1$ 条平行直线构成。否则 $\text{Vor}(P)$ 是连通图，其边要么是线段，要么是射线

如果不是所有点共线

证明 $\text{Vor}(P)$ 中边为线段或射线

反正如果有一条边是完整直线 e ， e 是 $V(p_i)$ 和 $V(p_j)$ 的共同边界

$\forall p_k \in p_i, p_j$ 不共线



p_j 与 p_k 间垂直平分线不能与 e 平行
 $\therefore p_j$ 与 e 相交。而 e 落在 $h(p_k, p_j)$
 内。及 a 不可能属于 $V(p_j)$ 的边界
 因为~~在~~ a 上任意到 p_k 距离
 比到 p_j 近。

2) 证 $\text{Vor}(P)$ 连通。

反证。设某个 $V(p_i)$ 将平面分成两个互不相交的部分
 \because 区域是凸集 $V(p_i)$ 只能是一个（两端无限窄）的多形区域
 边界是两条平行直线 与矛盾

规模：如果基点个数为n，则Voronoi图是平面图（假设所有射线都连到一个共同的无穷远点），有n个面，顶点个数最多 $2n-5$ ，边数最多 $3n-6$ 。

证明：

$$\begin{aligned} & \underline{(n_v+1)} - \underline{n_e} + \underline{n} = 2 \\ \therefore n_v &= n_e - n + 1 \text{ 或 } n_e = n_v + n - 1 \\ \text{加 } 3V_\infty \text{ 后, 每条边对应 } 2 \text{ 个顶点} \\ \therefore \text{所有面上度数之和} &\text{为边数的 } 2 \text{ 倍} \\ \therefore \text{包括 } V_\infty \text{ 在内之能度数不小于 } 3 \\ \therefore \underline{2n_e} &\geq 3(n_v+1) \\ 2n_e &\geq 3(n_e - n + 1) \\ n_e &\leq 3n - 6 \end{aligned}$$

求Voronoi图

一种简单的方法：通过求半平面 $h(p_i, p_j)$ ($i \neq j$)的交集求出每一个 $V(p_i)$ ，计算时间 $O(n^2 \log n)$

历史上，有很多年都使用一种 $O(n^2)$ 的算法，后来曾经有一种基于分治法的渐近上界为 $O(n \log n)$ 的算法，但该算法相当复杂，不容易理解。后来Steven Fortune在1978年设计出来一种简单的扫描线算法来求解Voronoi图，时间复杂性为 $O(n \log n)$ 。

扫描线算法的难点

对任意一种扫描线算法，其关键在于能够有效知道所有即将发生的事件。

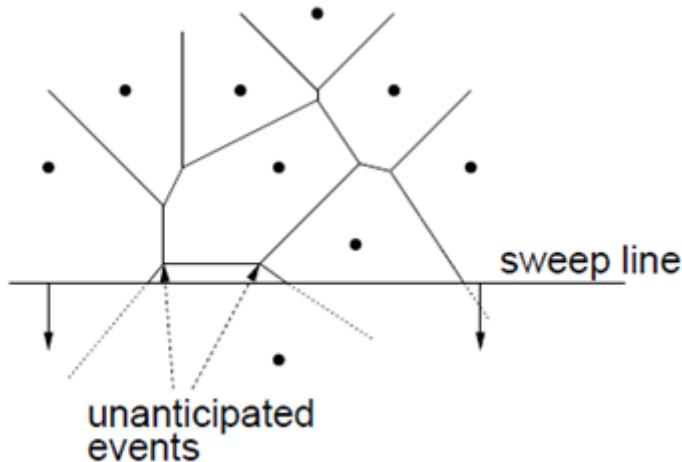
用扫描线算法求解Voronoi图的问题难点就在于如何预测下一事件何时何地发生

假设你设计了一种扫描线算法，在扫描线已经扫描过的部分，已经基于扫描过的点构造出了相应的Voronoi图。但是扫描线前方未扫描到的基点可能会生成一个位于扫描线后方的Voronoi顶点。

假设扫描线自上而下移动

则图中箭头所示两个顶点依赖于扫描线还未遇到的基点，因此算法未发现这两个顶点

如果让算法在遇到基点后再发现这两个顶点，那就太晚了。正是由于这种不可预测事件使扫描线算法的设计具有挑战性。



Fortune的独到之处

Q：扫描线后方哪些部分不会再发生变化？

A：将扫描线上方的半平面划分成两个区域，一个区域中的点距离扫描线上方某个基点的距离小于到扫描线的距离，另一个区域中的点离扫描线的距离则比到所有扫描线上方的基点的距离更近。

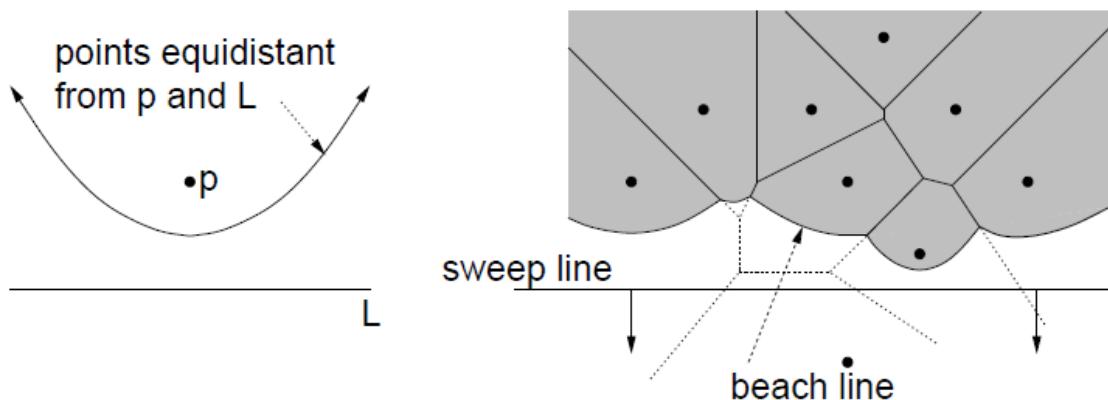
由两个对象控制扫描过程

第一个是一条水平扫描线自上而下移动

第二个是一条关于x轴单调的曲线，称为海岸线，由一系列抛物线构成，当扫描线向下移动时海岸线在其后面移动。

海岸线：是一个点集，其中的点距离扫描线上方最近基点的距离等于到扫描线的距离

对于海岸线上方的点 q ，距离其最近的基点不会再受扫描线下方基点的影响。因此海岸线上方的Voronoi图部分是“安全”的，因为计算其所需的信息已经全部获得了，无需任何扫描线下方基点的信息。



考虑海岸线上两条抛物线的交点，该点称为断点(breakpoint)，它离某两个基点以及扫描线的距离相等，也必然位于某条Voronoi边上。特别地，如果两条抛物线分别对应基点 p_i 和 p_j ，则该断点位于 p_i 和 p_j 之间的Voronoi边上。

引理：海岸线是一条关于x轴单调的曲线，由多条抛物线组成，其断点位于最终的Voronoi图的边上。

Fortune算法关注扫描线向下移动时海岸线的变化情况，尤其是跟踪断点的运动轨迹，因为断点是沿着Voronoi图的边移动的。

在计算过程中需要维护扫描线状态，以及一些离散的事件点，在这些事件点上，Voronoi图和海岸线的拓扑结构会发生变化。

扫描线状态：需要知道扫描线的当前位置（其y坐标），以及确定当前海岸线的基点（按从左往右的次序）。不需要存储海岸线的抛物线结构，这些抛物线的目的主要是用来理解算法的

事件：有两类事件

基点事件：扫描线经过一个新的基点时，一条新的抛物线会生成插入海岸线

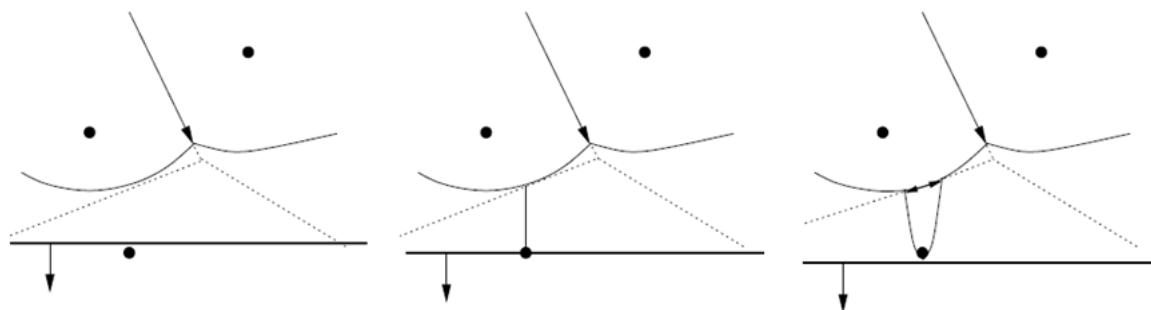
顶点事件：当一条抛物线的长度缩减为0时，该抛物线消失，在这一点会生成一个新的Voronoi顶点

基点事件

当扫描线经过一个基点时会产生一个基点事件

当扫描线遇到基点的一瞬间，其对应的抛物线是一条退化的垂直射线，方向是从基点到当前扫描线。当扫描线向下移动时，这条射线会展开成为海岸线上的一条抛物线

为处理基点事件，需要确定新基点正上方的抛物线，在其中插入一条新的无穷小的抛物线。当扫描线下移时，这条抛物线会展开，两个断点在向相反方向移动时共同得到一条Voronoi边，这条边一开始与扫描线上方Voronoi图的其它部分不相连，但随着扫描线的移动，这条边不断生长，最后会与其它边相遇，从而与Voronoi图的其它部分相连



引理：只有通过基点事件才能在海岸线上生成一条新的抛物线

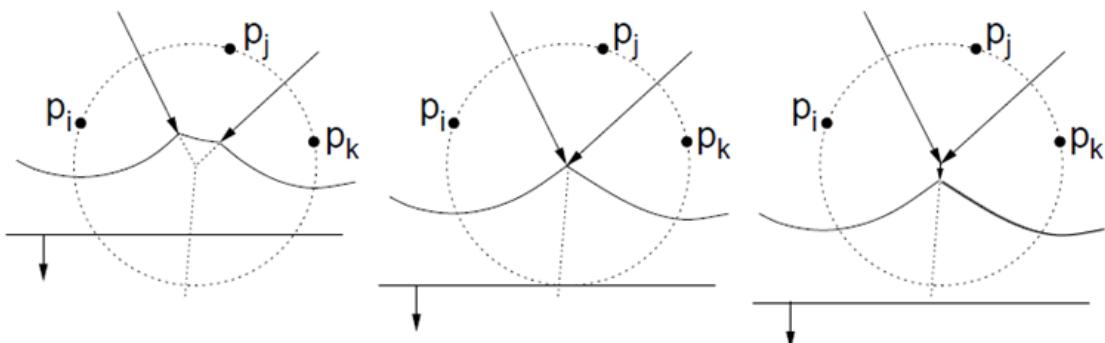
海岸线上的抛物线最多只有 $2n-1$ 条，因为每个基点会生成一条抛物线，并将已有的一条抛物线一分为二，从而除了第一个基点外，每个基点都可以使抛物线总数增加2

基点事件是事先都已经知道的，只要将基点按y坐标排序即可。

顶点事件

顶点事件是在算法运行过程中动态生成的。每个事件都由海岸线上相邻的抛物线/基点生成，3个基点/3条抛物线会生成一个顶点事件

考虑相邻的三个基点 p_i , p_j 和 p_k ，对应了海岸线上从左往右相邻的三条抛物线。这三个基点位于某个圆上，假设这个外接圆还有部分在当前扫描线下方，即Voronoi顶点还未生成，同时假设这个圆内也不会有基点位于扫描线下方，即不会有阻碍顶点的生成



考虑扫描线移动到恰好与外接圆下端相切的瞬间，此时圆心距离3个基点以及扫描线的距离相等，三条抛物线都经过圆心，也意味着中间的 p_j 对应的抛物线恰好在海岸线中消失。在Voronoi图中， (p_i,p_j) 的平分线和 (p_j,p_k) 的平分线相交，交点为Voronoi顶点，剩下 (p_i,p_k) 的平分线

引理：抛物线从海岸线中消失的唯一方式是海岸线遇到了顶点事件

基点事件会生成新的抛物线

顶点事件中抛物线会消失

基点事件后一条新的Voronoi边开始生长

顶点事件中两条生长的Voronoi边相遇形成Voronoi顶点

扫描线算法：数据结构

海岸线：用字典表示。不用显式地存储其中的抛物线，它们仅仅是用来导出算法的。只需存储当前海岸线对应的基点，但要注意的是海岸线中一个基点可能会对应多条抛物线，但一条海岸线中的抛物线的数量不会超过 $2n-1$ （参见基点事件中的图）

在相邻的两个基点 p_i 和 p_j 间存在断点。虽然断点会随着扫描线而移动，但是其位置可以通过 p_i 、 p_j 和扫描线计算而得，它是一个圆的圆心，该圆经过 p_i 和 p_j ，并且与扫描线相切。因此与海岸线一样，不需要显式地存储断点，只要在需要的时候进行计算

对海岸线需要的操作

(1) 给定扫描线的位置，求出与指定垂直线相交的抛物线。方法是通过对断点的二分搜索

(2) 求前驱和后续

(3) 在已有抛物线 p_j 中插入新的抛物线 p_i 。需要将 p_j 一分为二，从而得到三段抛物线 p_j ， p_i 和 p_j

(4) 删除抛物线

利用字典结构，这些操作都可以在 $O(\log n)$ 时间内解决

事件队列：利用优先队列，可以插入和删除事件。具有最大y坐标的事件会先出队列。对每个基点，在队列中只需存储其y坐标

对于海岸线连续的三段抛物线 p_i ， p_j 和 p_k ，计算其外接圆，如果圆上最下方的点在扫描线下方，则生成一个顶点事件，需要得到该点的y坐标并将其存在队列中。顶点事件与三段抛物线之间有相互链接，既可以通过顶点事件找到抛物线，也可以由抛物线找到顶点事件。

扫描线算法：事件

基点事件：设 p_i 为当前基点，垂直向上找到基点 p_j 对应的抛物线，用 p_j, p_i, p_j 代替海岸线中原来的 p_j 。同时将 p_i 和 p_j 间的平分线作为Voronoi图的边。原来一些与 p_j 相关的三元组对应的事件需要删除，新生成一些与 p_i 有关的三元组对应的事件。

假设在事件发生前海岸线序列为

$\langle p_1, p_2, p_j, p_3, p_4 \rangle$

则 p_i 将 p_j 分为 p'_j 和 p''_j ， p'_j 和 p''_j 实际上都对应原来的基点 p_j ，新的海岸线序列为

$\langle p_1, p, p'_j, p_i, p''_j, p_3, p_4 \rangle$

原来与 p_2, p_j, p_3 对应的事件需要删除，同时新生成 p_2, p'_j, p_i 对应的事件和 p_i, p''_j, p_3 对应的事件，但不需要生成 p'_j, p_i, p''_j 对应的事件，因为其中只涉及到两个不同的基点

顶点事件：设 p_i , p_j 和 p_k 是从左往右的三个产生该事件的基点。从海岸线中删除 p_j 对应的抛物线，在Voronoi图中新生成一个顶点作为平分线 (p_i, p_j) , (p_j, p_k) 对应的边的顶点，同时新生成平分线 (p_i, p_k) 对应的边，该边将向下生长。最后删除与 p_j 相关的事件，新生成的事件对应的连续三个基点中有两个是 p_i 和 p_k 。

假设事件发生前海岸线为

$\langle p_1, p_i, p_j, p_k, p_2 \rangle$

事件发生后海岸线为

$\langle p_1, p_i, p_k, p_2 \rangle$

删除与 p_1, p_i, p_j 和 p_j, p_k, p_2 相关的事件(与 p_i, p_j, p_k 相关的事件已被删除)，同时生成 p_1, p_i, p_k 和 p_i, p_k, p_2 对应的事件

每个事件本身处理时间 $O(1)$ ，加上访问常数个数据结构，访问时间 $O(\log n)$ ，数据结构规模 $O(n)$

时间复杂性 $O(n \log n)$ ，空间复杂性 $O(n)$

Ch 09 代数和数值算法

数值算法是干什么的

例如有一阶微分方程初值问题

$$\begin{cases} \frac{dy}{dx} = 2x \\ y(0) = 1 \end{cases}$$

数学问题：求出函数解析表达式 $y=y(x)$

数值问题：求出函数 $y=y(x)$ 在某些点 x 的近似函数值

数值算法的特点

计算复杂性：

时间，空间都尽量小

当输入的规模增长时，一些适用于小输入的算法将变得不再高效

什么是输入规模

计算可靠性：

收敛性/可行性

存在误差，运算结果不能产生太大的偏差并且能够控制误差

稳定性，即初始数据产生的误差对结果的影响

误差的来源与分类

模型误差：从实际问题抽象出数学模型

观测误差：通过观测得到模型中某些参数（或物理量）的值

截断误差：数学模型与数值算法之间的误差求近似解

舍入误差：由于机器字长有限，原始数据和计算过程会产生新的误差

计算 $\int_0^1 e^{-x^2} dx = 0.747\dots$

方法：Taylor展开再积分

$$\int_0^1 e^{-x^2} dx = \int_0^1 \left(1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!} - \dots\right) dx = \underbrace{1 - \frac{1}{3} + \frac{1}{2!} * \frac{1}{5} - \frac{1}{3!} * \frac{1}{7} + \frac{1}{4!} * \frac{1}{9} - \dots}_{S_4}$$

取 $\int_0^1 e^{-x^2} dx \approx S_4$

R_4 称为截断误差，由截去部分引起 $|R_4| < \frac{1}{4!} * \frac{1}{9} < 0.005$

同时 $S_4 = 1 - \frac{1}{3} + \frac{1}{10} - \frac{1}{42} \approx 1 - 0.333 + 0.1 - 0.024 = 0.743$

舍入误差 $< 0.0005 * 2 = 0.001$ ，由留下部分引起

总体误差 $< 0.005 + 0.001 = 0.006$

大数吃小数

计算： $f(i) = 2^{i+1} - 2^i$

这似乎是个很幼稚的问题，结果显然应该是1。但是如果在计算机上进行计算的话，只要i比较大（在我的计算机上， $i > 50$ ），就会给出0这样一个奇怪的结果。

这是因为在计算机内 2^i 存储为 $(0.1)_2 \times 2^{i+1}$ ，其中 $(0.1)_2$ 表示二进制的0.1，即十进制的0.5，而1存储为 $(0.1)_2 \times 2^1$ 。做加法时，两加数的指数部分先向大数对齐，再将尾数部分相加，即1的指数部分必须变为 2^{i+1} ，这时 $1 = \underbrace{(0.00\dots0001)_2}_{i \uparrow 0} \times 2^{i+1}$

当i比较大时，这通常都超出了浮点数尾数的位数限制，部分尾数被舍弃，使得1变成了0，从而有 $2^{i+1} = 2^i$ 。这种现象被称为大数吃小数。

稳定性

计算 $I_n = \frac{1}{e} \int_0^1 x^n e^x dx$

首先根据分部积分法得到递推关系式

$$I_n = \frac{1}{e} \left[x^n e^x \Big|_0^1 - n \int_0^1 x^{n-1} e^x dx \right] = 1 - n I_{n-1}$$

其初值条件为

$$I_0 = \frac{1}{e} \int_0^1 e^x dx = 1 - \frac{1}{e}$$

有了递推关系式和初值条件，相应的计算就应该非常容易了。

假设所使用的计算机有8位有效数字，则 $I(0) \approx 0.63212056$ ，记为 I_0^* ，在此基础上，可以得到：

$$I_1^* = 1 - 1 * I_0^* \approx 0.36787944$$

.....

$$I_{10}^* = 1 - 10 * I_9^* \approx 0.08812800$$

$$I_{11}^* = 1 - 11 * I_{10}^* \approx 0.03059200$$

$$I_{12}^* = 1 - 12 * I_{11}^* \approx 0.63289600$$

$$I_{13}^* = 1 - 13 * I_{12}^* \approx -7.2276480$$

$$I_{14}^* = 1 - 14 * I_{13}^* \approx 94.959424$$

$$I_{15}^* = 1 - 15 * I_{14}^* \approx -1423.3914$$



看上去一切正常...吗？

由于

$$\frac{1}{e} \int_0^1 x^n e^0 dx < I_n < \frac{1}{e} \int_0^1 x^n e^1 dx$$

所以

$$0 < \frac{1}{e(n+1)} < I_n < \frac{1}{n+1} < 1$$

那么，在使用正确的计算公式的情况下，为什么得到了离谱的计算结果呢？

考虑第n步的误差

$$|E_n| = |I_n - I_n^*| = |(1 - n * I_{n-1}) - (1 - n * I_{n-1}^*)| = n |E_{n-1}| = \dots = n! |E_0|$$

虽然初始误差 $|E_0| < 0.5 * 10^{-8}$, 但是误差随着计算过程的进行迅速积累, 从而得到了荒谬的结果。

造成这种情况的是不稳定的算法!

由

$$I_n = 1 - n * I_{n-1}$$

可以得到等价的公式

$$I_{n-1} = (1 - I_n) / n$$

这个式子意味着由 I_n 求 I_{n-1} , 感觉比较奇怪, 毕竟我们所知道的初始条件是 I_0^* , 怎么能够倒过来计算呢?

根据 $\frac{1}{e(N+1)} < I_N < \frac{1}{N+1}$

我们可以得到的估计值

$$I_N^* = \frac{1}{2} \left[\frac{1}{e(N+1)} + \frac{1}{N+1} \right]$$

再反推要求的 I_n ($n \ll N$)。虽然 I_N^* 肯定存在误差, 但此时误差的递推关系式为

$$|E_{N-1}| = \left| \frac{1}{N} (1 - I_N) - \frac{1}{N} (1 - I_N^*) \right| = \frac{1}{N} |E_N|$$

从而对于 $n < N$, 有 $|E_n| = \frac{1}{N(N-1)\dots(n+1)} |E_N| = \frac{n!}{N!} |E_N|$

即误差是逐步递减的, 从存在较大误差的开始计算得到的却误差很小, 这样的算法称为稳定的算法。

取

$$I_{15}^* = (1/e(15+1) + 1/(15+1))/2 \approx 0.042746233$$

则

$$I_{14}^* = (1 - I_{15}^*)/15 \approx 0.063816918$$

$$I_{13}^* = (1 - I_{14}^*)/14 \approx 0.066870220$$

$$I_{12}^* = (1 - I_{13}^*)/13 \approx 0.071779214$$

$$I_{11}^* = (1 - I_{12}^*)/12 \approx 0.077351732$$

...

$$I_1^* = (1 - I_2^*)/2 \approx 0.36787944$$

$$I_0^* = (1 - I_1^*)/1 \approx 0.63212056$$

对某些问题本身如果输入数据有微小扰动，就会引起输出数据（即问题真解）的很大扰动，这就是病态问题。它是问题本身性质所决定的，与算法无关，也就是说对病态问题，用任何算法直接计算都将产生不稳定性。

求幂运算

问题：给定两个正整数n和k，计算 n^k

直接的归纳法： $n^k = nn^{k-1}$

Algorithm Power(n,k)

Input: n and k (two positive integers)

Output: P (the value of n^k)

begin

 P:=n;

 for i:=1 to k-1 do

 P:=n*P

end

由于所减小的是k的值，而不是其规模，所以直接算法需要k次迭代。又因为k的规模是 $\log_2 k$ ，所以迭代的次数是k规模的指数函数($k=2^{\lceil \log_2 k \rceil}$)。

另一种归纳方法： $n^k = (n^{k/2})^2$ ，将问题归约成输入参数为n和k/2的问题。将k的值减小一半对应于将其规模减小一个常数值，故乘法次数和k的规模成线性关系。

如果k是偶数，就简单地对参数为k/2的问题的解进行平方；如果k是奇数，则将参数为(k-1)/2的问题的解平方后再乘以n。所以，乘法的运算量至多为 $2\log_2 k$ 。

```
Algorithm Power_by_Repeated_Squaring(n,k)
Input: n and k (two positive integers)
Output: P (the value of  $n^k$ )
begin
    if k=1 then P:=n;
    else
        z:=Power_by_Repeated_Squaring(n, k div 2);
        if k mod 2=0 then
            P:=z*z
        else
            P:=n*z*z
    end
```

16

注意：乘法运算的数目为 $O(\log k)$ ，但数值会随着算法的进行变得越来越大，因而乘法运算的开销也将不断变大。

最大公约数

两个正整数m和n的最大公约数GCD是指满足以下两个条件的唯一正整数k：(1)n和m都能被k整除；(2)能同时整除m和n的其他整数均小于k，一般用GCD(m,n)来表示m和n的最大公约数。

问题：寻找两个给定正整数m和n的最大公约数。

关键：如果k能同时整除m和n，那么它能整除后两者的差

问题：如果 $n > m$ ，那么 $\text{GCD}(n,m) = \text{GCD}(n-m,m)$ ，由此可得一个更小的问题。但是，减小的仍然是问题中数的值，而不是规模。

方法：可重复从n中减去m直至结果小于m。但这恰好等价于n除以m并得到余数的过程，而除法运算可以很快完成。

```
Algorithm GCD(m,n)
Input: m and n (two positive integers)
Output: gcd (the gcd of m and n)
begin
    a:=max(n,m);
    b:=min(n,m);
    r:=1;
    while r>0 do {r is the remainder}
        r:=a mod b;
        a:=b;
        b:=r;
    gcd:=a
end
```

a	b	r
a	b	$a \% b$
b	$a \% b$	$b \% (a \% b)$
$a \% b$	
a	$a \% b$	$< \frac{a}{2}$

复杂性：算法的运行时间关于 $m+n$ 的规模是线性的，即它的运行时间(每步运算的花费与操作数的规模无关，均计为一步)是 $O(\log(n+m))$ 。

最小公倍数： $m \times n / GCD(m, n)$

多项式乘法

问题：给定 $n-1$ 次多项式 $P=p_{n-1}x^{n-1}+\dots+p_0$ 和 $Q=q_{n-1}x^{n-1}+\dots+q_0$ ，计算 P 和 Q 的乘积。

直接方法：利用 $PQ=(p_{n-1}x^{n-1}+\dots+p_0)(q_{n-1}x^{n-1}+\dots+q_0)=p_{n-1}q_{n-1}x^{2n-2}+\dots+(p_{n-1}q_{i+1}+p_{n-2}q_{i+2}+\dots+p_{i+1}q_{n-1})x^{n+i}+\dots+p_0q_0$ 。

乘法和加法的运算量是 $O(n^2)$

分治算法：假定 n 是2的幂，将每个多项式划分成大小相等的两部分，设 $P=P_1+x^{n/2}P_2$ 而 $Q=Q_1+x^{n/2}Q_2$ ，其中

$$\begin{aligned} P_1 &= p_0 + p_1x + \dots + p_{n/2-1}x^{n/2-1} & P_2 &= p_{n/2} + p_{n/2+1}x + \dots + p_{n-1}x^{n/2-1} \\ Q_1 &= q_0 + q_1x + \dots + q_{n/2-1}x^{n/2-1} & Q_2 &= q_{n/2} + q_{n/2+1}x + \dots + q_{n-1}x^{n/2-1}。 \end{aligned}$$

可得

$$PQ = (P_1 + P_2x^{n/2})(Q_1 + Q_2x^{n/2}) = P_1Q_1 + (P_1Q_2 + P_2Q_1)x^{n/2} + P_2Q_2x^n。$$

运算量 $T(n)=4T(n/2)+O(n)$ ， $T(1)=1$ ，解是 $O(n^2)$ 。

19

设 $A=P_1*Q_1$, $B=P_2*Q_1$, $C=P_1*Q_2$, $D=P_2*Q_2$, 要计算

$A+(B+C)x^{n/2}+Dx^n$ ，通过观察可以发现，不需要分别计算 B 和 C ，只需要得到它们的和即可！

计算乘积 $E=(P_1+P_2)(Q_1+Q_2)$ ，那么 $B+C=E-A-D$ 。因此，只需要计算3个小规模的多项式乘积，即 A 、 D 和 E ，其余操作可以通过加法和减法运算进行计算

新递归关系为 $T(n)=3T(n/2)+O(n)$ ，从而可以推得

$$T(n)=O(n^{\log_2 3})=O(n^{1.59})。$$

正整数乘法

设计快速算法，实现两个n位正整数相乘 $O(n^2)$

设计快速算法，实现两个n位正整数相乘

$O(n^2)$

$O(n^{1.59})$ 1960 Karatsuba

$O(n \cdot \log n \cdot \log(\log n))$ 引入FFT 1971 Arnold Schonhage和Volker Strassen

$O(n \cdot \log n \cdot K^{\log n})$ 2007 Martin Fürer K为大于1的常数，最新的K=4

$O(n \log n)$ 2019 Harvey和van der Hoeven

10214857091104455251940635045059417341952位以上的二进制数字

$$\begin{aligned} 1234 &= 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \\ &\quad \times 10^0 \\ P &= 1 \cdot x^3 + 2 \cdot x^2 + 3 \cdot x^1 + 4 \\ P(10) &= 1234 \end{aligned}$$

矩阵乘法

问题：计算两个 $n \times n$ 实数矩阵A和B的乘积C=A×B。

直接方法：按照矩阵乘法定义，需要使用 n^3 次乘法运算和 $(n-1)$

n^2 次加法运算。注意到这里的n代表的是矩阵行和列的个数，而不是输入的规模 n^2 。

Winograd算法：设n是偶数。记

$$A_i = \sum_{k=1}^{n/2} a_{i,2k-1} \cdot a_{i,2k} \quad B_j = \sum_{k=1}^{n/2} b_{2k-1,j} \cdot b_{2k,j}$$

可得

$$c_{i,j} = \sum_{k=1}^{n/2} (a_{i,2k-1} + b_{2k,j}) \cdot (a_{i,2k} + b_{2k-1,j}) - A_i - B_j$$

由于对于每一行每一列， A_i 和 B_j 只需要计算一次，故计算所有的 A_i 和 B_j 只需 n^2 次乘法运算。因此，总共的乘法运算量减少至 $n^3/2+n^2$ ，而加法运算量则大约增加了 $n^3/2$ 。从而在加法运算快于乘法运算的情况下，该算法要优于直接算法。

假设 $n=2^k$, $k \geq 0$ 。如果 $n \geq 2$, 则 A, B, C 可以划分为如下形式

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

利用分治法求解 C 的过程可以用如下形式表示:

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

此递归形式需要 n^3 次乘法和 $n^3 - n^2$ 次加法, 与直接计算的计算量一致。

Strassen 算法

令

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

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

计算下列乘积

$$\begin{aligned} d_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ d_2 &= (a_{21} + a_{22})b_{11} \\ d_3 &= a_{11}(b_{12} - b_{22}) \\ d_4 &= a_{22}(b_{21} - b_{11}) \\ d_5 &= (a_{11} + a_{12})b_{22} \\ d_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ d_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

则 C 为

$$C = \begin{pmatrix} d_1 + d_4 - d_5 + d_7 & d_3 + d_5 \\ d_2 + d_4 & d_1 + d_3 - d_2 + d_6 \end{pmatrix}$$

$$\begin{aligned} T(n) &= \begin{cases} 1m & \text{if } n = 1 \\ 7T(n/2) + 18(n/2)^2 a & \text{if } n \geq 2 \end{cases} \\ &= n^{\log 7} m + 6n^{\log 7} a - 6n^2 a \\ \text{计算时间为 } \Theta(n^{\log 7}) &= O(n^{2.81}) \end{aligned}$$

26

Strassen 算法有 3 个主要的缺点:

1. 根据经验研究发现, n 至少需要 100 才能使 Strassen 算法快于 $O(n^3)$ 的直接算法。
2. Strassen 算法不如直接算法稳定, 即对于相同的输入误差, Strassen 算法很可能产生更大的输出误差。
3. Strassen 算法显然比直接算法复杂得多且更难实现, 此外, Strassen 算法不像常规算法那样容易并行化。

目前已知矩阵乘法最佳算法为 $O(n^{2.376})$.

布尔矩阵

问题：计算两个 $n \times n$ 布尔矩阵的乘积

为整数所设计的算法通常不能用于布尔类型。布尔和定义中产生的一个问题是在减法运算无法定义(0+1和1+1都定义为1，因此1-1既可以是1，也可以是0)，然而Strassen算法却需要减法运算，所以不适用于布尔矩阵。

技巧1：环

技巧2：将k个操作数存储在一个大小为k的计算机字中。矩阵乘法的常规算法由 n^2 个“行乘以列”的乘积(或内积)所组成，假定k可整除n，因而可以将每个内积分解成 n/k 个k维布尔向量的内积的总和。

第一个思想

预先计算所有可能的k维布尔内积：由于其中涉及两个k维布尔向量，所以共有 2^{2k} 种可能，可以在时间 $O(k2^{2k})$ 内计算，并将其存储在规模为 $2^k \times 2^k$ 比特的二维表中。两向量a和b的内积存储在条目 (i_a, i_b) ， i_a 和 i_b 分别指a和b中的k比特所代表的整数。这样，给定两个k维布尔向量，可以通过简单检索该表计算内积。如果可以在 $O(1)$ 时间内访问规模为 2^{2k} 的表，那么在表构造后每个k维内积就可以在常数时间内计算得出。

例如，设 $k=\lfloor \log_2 n/2 \rfloor$ ，表的规模此时为 $O(n)$ ，构造该表本身需要 $O(n \log n)$ 时间。假定可以在常数时间内访问规模为 $O(n)$ 的表。一旦构造了该表，就可以在时间 $O(n/k)=O(n/\log n)$ 内计算n维布尔内积。注意到该表所依赖的是k的值，而不是矩阵。因此，计算两布尔矩阵的乘积可以在时间 $O(n^3/\log n)$ 以及额外空间 $O(n)$ 内完成。也可以选取k为 $\lfloor \log_2 n \rfloor$ ，此时表的规模为 $O(n^2)$ ，而矩阵乘法时间可以减半。

第二个思想

考虑两个 $n \times n$ 的布尔矩阵A和B，矩阵乘法通过如下“**A列乘以B行**”的方式计算：记A的第r列为 $A_C[r]$ ，而B的第r行为 $B_R[r]$ ； $A_C[r]$ 可以视为一个 $n \times 1$ 矩阵，而 $B_R[r]$ 则可以视为一个 $1 \times n$ 矩阵。 $A_C[r]$ 和 $B_R[r]$ 的乘积是一个 $n \times n$ 矩阵，后者的第ij个元素是 $A_C[r]$ 的第i个元素与 $B_R[r]$ 的第j个元素的乘积。显然，有 $A \times B = \sum A_C[r]B_R[r]$

现在将A的列、B的行划分成 n/k 个大小相等的组(假定 n/k 是整数)，即将A分解成 $A_1, A_2, \dots, A_{n/k}$ ，使得每一个 A_i 构成大小为 $n \times k$ 的矩阵；同理，将B分解成 $B_1, B_2, \dots, B_{n/k}$ ，使得每一个 B_i 构成大小为 $k \times n$ 的矩阵。显然，有 $A \times B = \sum A_i B_i$

设 $C_i = A_i B_i$, 其第j行是若干 B_i 行的布尔和(根据 A_i 的第j行)。这里并不直接计算 C_i 的每一行，而是使用类似于前述算法中所使用的方法，即预先计算所有的可能性。

由于 A_i 每行中有 k 个元素，所以 B_i 行的各种可能组合一共有 2^k 种。

令 $k = \log_2 n$ 并再一次假定 k 是一个整数，预先计算所有 $2^k = 2^{\lceil \log_2 n \rceil} = n$ 种组合，然后将结果存储在一个表中。

相比于先前的算法，该表包含 n 行而不是 n 比特；因此，存储空间需要 $O(n^2)$ 。此外，由于该表还依赖于 B_i ，所以必须为每一个 B_i 均建立起与之对应的表。

为了寻找 C_i 的第 j 行，检索 A_i 的第 j 行并找到需要相加的 B_i 行组合。 A_i 第 j 行的二进制表示所对应的整数可以代表这样的组合，这个整数是 C_i 第 j 行在表中的存储地址。在表中寻找到 C_i 中的一行需要花费 $O(1)$ 时间，而将这一行复制到 C_i 中适合位置需要花费 $O(n)$ 时间。

综上，可以在时间 $O(n^2)$ 内计算 C_i 。

$$A_i = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \quad B_i = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$$C_i = \begin{pmatrix} 0 & 1 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 1 \end{pmatrix} \quad \begin{matrix} B_i(3) \\ 2+3 \\ 1+2 \end{matrix}$$

001->3

011->2+3

100->1

110->1+2

计算 B_i 行之和的各种可能组合需要多少时间？ $O(n2^k)$

设每种行的组合对应于一个整数

归纳基础：计算对应于0的行之和是显然可行的。

归纳假设：知道如何计算对应小于i的整数的行之和组合。

首先，假定*i-1*的二进制表示为xxxx011111(即最低位的0后继j个1)，那么对应于*i*的行之和等于对应于xxxx000000的行之和加上对应于0000100000的行之和。由于xxxx000000小于*i*，根据归纳假设可以得到其所对应的行之和，现在只需要再添加一行即可，添加一行需要进行n次布尔加法运算，时间为 $O(n)$ 。总共有 2^k 种组合，因此所有的预先计算均可以通过 $O(n2^k)$ 次运算完成。若 $k=\log_2 n$ ，则运行时间为 $O(n^2)$ 。

```
Algorithm Boolean_Matrix_Multiplication(A, B, n, k)
Input: A, B (two n*n Boolean matrices), and k (an integer, and k divides n for
simplicity)
Output: C (the product of A and B)
begin
    Initialize the matrix C to 0
    for i:=0 to n/k-1 do
        Construct Tablei;
        {Tablei is an  $2^k$  array of Boolean vectors of size n which contains all
possible combinations of sums of k rows of Bi}
        m:=i*k;
        for j=1 to n do
            Let Addr be the k-bit number A[j,m+1]A[j,m+2]...A[j,m+k];
            add Tablei[Addr] to row j in C
end
```

复杂性：为了计算 AB ，必须计算 n/k 次 A_iB_i 的乘积。由于每一个这样的乘积要花费 $O(n^2)$ 时间，外加构造表所花费的 $O(n2^k)$ ，该算法总共的运行时间为 $O(n^3/k+n^22^k/k)$ 。如果 $k=\log_2 n$ ，那么运行时间为 $(n^3/\log n)$ 。

矩阵链相乘

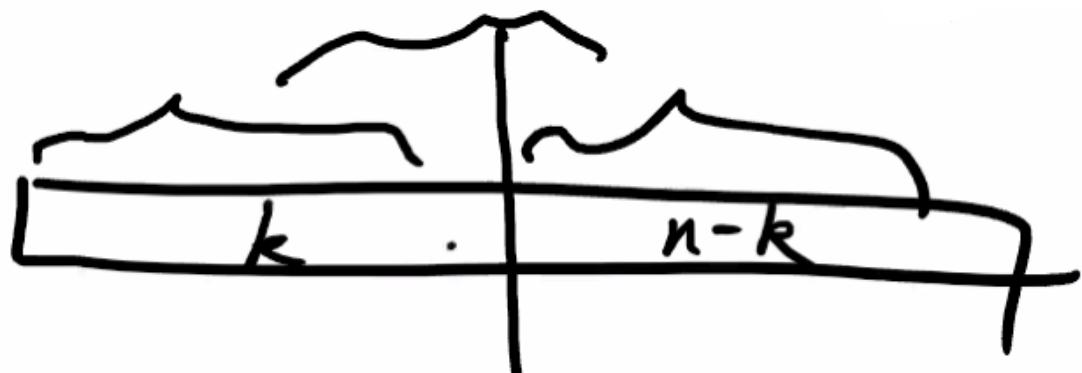
n 个矩阵 $M_1M_2\dots M_n$ 相乘的代价依赖于乘法的次序

蛮力搜索法：尝试所有可能的计算次序

运行时间 $\Omega(4^n / \sqrt{n})$

$$\begin{array}{c}
 \overline{A_{m \times n} \quad B_{n \times p} \quad \underline{m \times n \times p} \quad C} \\
 \overline{M_1 \quad 2 \times 10 \quad M_2 \quad 10 \times 2 \quad M_3 \quad 2 \times 10} \\
 \\
 \underbrace{(M_1 M_2) \cdot M_3}_{2 \times 2} \quad \underbrace{2 \times 10 \times 2 + 2 \times 2 \times 10}_{80}
 \end{array}$$

$$M_1 (M_2 M_3)_{10 \times 10} \quad \underbrace{10 \times 2 \times 10 + 2 \times 10 \times 10}_{(400)}$$



$$f(n) = \sum_{k=1}^{n-1} f(k) \cdot f(n-k)$$

$$f(2) = 1 \quad f(1) = 1$$

$$f(3) = 2$$

$$f(n) = \frac{1}{n} \binom{n-1}{2n-2} = \frac{1}{n} \frac{(2n-2)!}{((n-1)!)^2}$$

$$n! \approx \sqrt{2\pi n} \cdot (\frac{1}{e})^n$$

$$f(n) = \frac{4^n}{4\sqrt{\pi} \cdot n^{1.5}} \approx \sqrt{\left(\frac{4}{n^{1.5}}\right)}$$

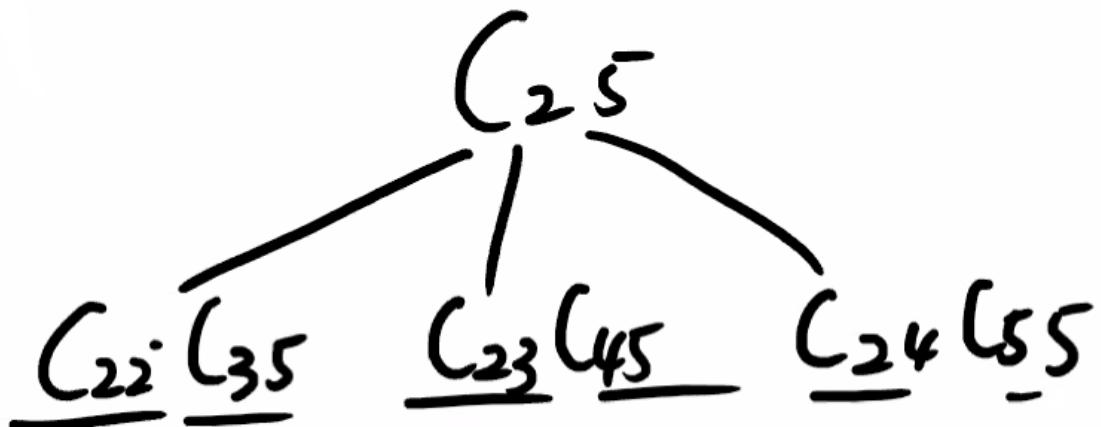
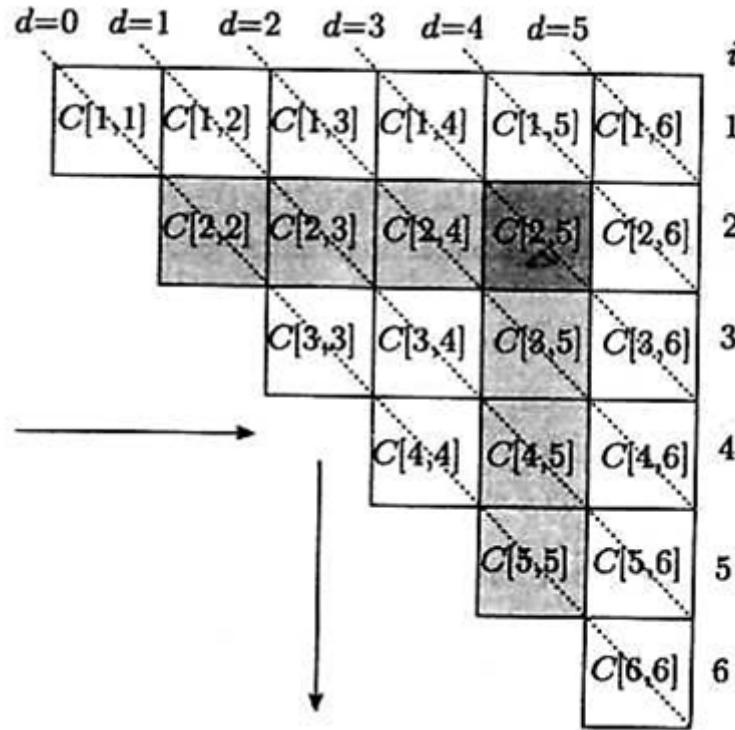
假设给定 $n+1$ 个正整数 r_1, r_2, \dots, r_{n+1} , 其中 r_i 和 r_{i+1} 为矩阵 M_i 的行数和列数, $1 \leq i \leq n$.

记 M_{ij} 为 $M_i M_{i+1} \dots M_j$ 的乘积, $C[i, j]$ 表示计算 M_{ij} 所需要的最少乘法数量, 则

$$C[i, j] = \min_{i < k \leq j} \{C[i, k-1] + C[k, j] + r_i r_k r_{j+1}\}$$

特别地:

$$C[1, n] = \min_{1 < k \leq n} \{C[1, k-1] + C[k, n] + r_1 r_k r_{n+1}\}$$



最终结果在右上角，按照从左到右斜线的顺序进行计算

```

Algorithm MATCHAIN
Input: An array r[1..n+1] of positive integers corresponding to the dimensions
of a chain of n matrices, where r[1..n] are the number of rows in the n matrices
and r[n+1] is the number of columns in Mn
Output: The least number of scalar multiplications required to multiply the n
matrices
1. for i<-1 to n      {Fill in diagonal d0}
2.   c[i,i]<-0
3. end for
4. for d<-1 to n-1    {Fill in diagonals d1 to dn-1}
5.   for i<-1 to n-d   {Fill in entries in diagonal di}
6.     j<-i+d
7.     comment: The next three lines computes c[i,j]
8.     c[i,j]<-+inf
9.     for k<-i+1 to j
10.       c[i,j]<-min{c[i,j],c[i,k-1]+c[k,j]+r[i]r[k]r[j+1]}
11.     end for
12.   end for

```

```

13. end for
14. return c[1,n]

```

$$\begin{aligned}
T(n) &= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \left(\sum_{k=i+1}^{i+d} 1 \right) \\
&= \sum_{d=1}^{n-1} \left(\sum_i d \right) = \sum_{d=1}^{n-1} d(n-d) \\
&= n \sum d - \sum d^2 = \frac{n^3 - n}{6}
\end{aligned}$$

Time: $\Theta(n^3)$

Space: $\Theta(n^2)$

多项式乘法(2)

问题：给定两个多项式 $p(x)$ 和 $q(x)$ ，计算它们的乘积 $p(x)q(x)$

多项式通常的表示方式是将多项式按照次数递减的系数列表方式表示成 $P = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$ ，也可用 n 个不同的点表示多项式

通过多项式求值的方法，可以从系数的表示转变成点的表示方式，根据 Horner 规则可以使用 n 次乘法运算计算多项式在任意一点的值。由于需要在任意 n 个点对 $p(x)$ 进行求值，所以总共需要 n^2 次乘法运算。从点的表示方式转变成系数表示方式的过程称为插值，其通常需要 $O(n^2)$ 次运算。

关键思想：不需要针对任意 n 个点，可以自由选取想要的任何 n 个不同点所组成的集合。

用 n 个不同的点表示多项式，即 n 个平面上的点可以表示唯一的 P 。

可以选择特殊的 n 个点来降低计算量

正向傅里叶变换

假定在n个不同点对任意一个n-1次多项式 $P=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\dots+a_1x+a_0$ 进行求值，并设n是2的幂，则上述多项式在点 x_0, x_1, \dots, x_{n-1} 的取值可以表示成以下矩阵和向量的乘法运算

$$\begin{bmatrix} 1 & x_0 & \cdots & x_0^{n-1} \\ 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{bmatrix}$$

考察任意两行第i行和第j行，由于点两两不等，不能令 $x_i=x_j$ ，但可以令 $x_j=-x_i$ ，从而有 $(x_i)^2=(x_j)^2$ 。由于 x_i 和 x_j 的所有偶数次幂均相等，所以可以节省一半涉及第i, j行的乘法运算。此外，对于其他每一行对都可以使用相同的方法，就能够将问题的规模减小一半。

17

$$\begin{bmatrix} 1 & x_0 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & \cdots & (x_1)^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n/2-1} & \cdots & (x_{n/2-1})^{n-1} \\ 1 & -x_0 & \cdots & (-x_0)^{n-1} \\ 1 & -x_1 & \cdots & (-x_1)^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & -x_{n/2-1} & \cdots & (-x_{n/2-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(-x_{n/2-1}) \end{bmatrix}$$

式中的 $n \times n$ 矩阵分成两个大小为 $n/2 \times n$ 的子矩阵，这两个矩阵非常相似。对于任何满足 $0 \leq i < n/2$ 的i，均有 $x_i = -x_{n/2+i}$ 。由于两个矩阵中偶次幂的系数恰好完全相等，它们只需要计算一次；而奇次幂的系数虽然不等，但是它们恰好互为相反数！

对于 $0 \leq i < n/2$, 将 $P(x_i)$ 和 $P(-x_i)$ 的表达式根据其偶次幂系数和奇次幂系数写成:

$$P(x) = E + O = \sum_{i=0}^{n/2-1} a_{2i} x^{2i} + \sum_{i=0}^{n/2+1} a_{2i+1} x^{2i+1}$$

其中

$$E = \sum_{i=0}^{n/2-1} a_{2i} (x^2)^i = P_e(x^2) \quad O = x \sum_{i=0}^{n/2-1} a_{2i+1} (x^2)^i = xP_o(x^2)$$

即

$$P(x) = P_e(x^2) + xP_o(x^2)$$

且有

$$P(-x) = P_e(x^2) + (-x)P_o(x^2)$$

为了求得n个点的值, 需要对 $0 \leq i < n/2$ 计算 $P(x_i)$ 和 $P(-x_i)$ 。为此只需对 $P_e(x^2)$ 和 $P_o(x^2)$ 各计算 $n/2$ 个值, 并执行 $n/2$ 次加法运算、 $n/2$ 次减法运算以及n次乘法运算。因此, 原问题变为两个规模为 $n/2$ 的子问题以及 $O(n)$ 次额外计算。

如果能的话, 就可以到递归关系 $T(n)=2T(n/2)+O(n)$, 从而得到一个 $O(n\log n)$ 的算法。但 $P(x)$ 中x的取值可以任意选择, 而 x^2 的值却只能取正数。

抽取对应计算 $P_e((x_i)^2)$ 的矩阵:

$$\begin{bmatrix} 1 & (x_0)^2 & (x_0)^4 & \cdots & (x_0)^{n-2} \\ 1 & (x_1)^2 & (x_1)^4 & \cdots & (x_1)^{n-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & (x_{n/2-1})^2 & (x_{n/2-1})^4 & \cdots & (x_{n/2-1})^{n-2} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} P_e(x_0) \\ P_e(x_1) \\ \vdots \\ P_e(x_{n/2-1}) \end{bmatrix}$$

如果对此子问题使用相同的技巧, 需要令 $(x_{n/4})^2 = -(x_0)^2$, 因而引入复数, 对于 $0 \leq j < n/4$, 令 $x_{j+n/4} = \sqrt{-1}x_j$, 这样又能再次将问题分解成两部分, 同时这次分解具有与第一次分解一样的性质, 只要解决两个规模为 $n/4$ 的子问题并进行 $O(n)$ 次额外计算, 就能够解决规模为 $n/2$ 的问题。

如果要将该过程进一步推广, 就需要一个等于 $\sqrt{\sqrt{-1}}$ 的数, 即满足“ $z^8=1$ 并且对所有 $0 < j < 8$ 均有 $z^j \neq 1$ ”的数z(从而可以推出 $z^4 = -1$ 以及 $z^2 = \sqrt{-1}$)。

n次单位原根(primitive nth root of unity), 用符号 ω 表示。 ω 满足如下条件:

$\omega^n=1$ 且 $\omega^j \neq 1$ (对于 $0 < j < n$)。

分别选取 $1, \omega, \omega^2, \dots, \omega^{n-1}$ 作为这n个点 x_0, x_1, \dots, x_{n-1} 。计算下列乘积:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

称这样的乘积为 $(a_0, a_1, \dots, a_{n-1})$ 的傅立叶变换。

对所有满足 $0 \leq j < n/2$ 的j, 均有 $x_{j+n/2} = \omega^{n/2}x_j = -x_j$, 故应用于规模为n的问题的归约有效。此外归约所生成的子问题有 $n/2$ 个点, 它们分别是 $1, \omega^2, \omega^4, \dots, \omega^{n-2}$, 而这恰好就是用 ω^2 替换 ω 之后规模为 $n/2$ 的子问题。由于 ω^2 是一个 $n/2$ 次单位原根。因此, 可以继续递归操作, 从而该算法的复杂性为 $O(n \log n)$ 。

Algorithm Fast_Fourier_Transform($n, a_0, a_1, a_{n-1}, \omega, varV$)

Input: n (an integer which is a power of 2), a_0, \dots, a_{n-1} (a sequence of elements whose type depends on the application), and ω (a primitive nth root of unity)

Output: V (an array in the range [0,n-1] of output elements)

begin

 if $n=1$ then

$V[0]:=a_0;$

 else

 Fast_Fourier_Transform($n/2, a_0, a_2, \dots, a_{n-2}, \omega^2, U$);

 Fast_Fourier_Transform($n/2, a_1, a_3, \dots, a_{n-1}, \omega^2, W$);

 for $j:=0$ to $n/2-1$ do

$V[j]:=U[j]+\omega^jW[j];$

$V[j+n/2]:=U[j]-\omega^jW[j];$

 end

逆向傅里叶变换

对两个给定多项式能够在点 $1, \omega, \dots, \omega^{n-1}$ 处利用快速傅立叶变换进行求值, 然后将得到的结果相乘, 从而得到乘积多项式在那些点的取值。但是还需要根据那些点的取值对乘积多项式的系数进行插值, 插值问题非常类似于求值问题, 可以用几乎完全相同的算法解决。

当给定多项式的系数(a_0, a_1, \dots, a_{n-1})并且想要计算多项式在n个点 $1, \omega, \omega^2, \dots, \omega^{n-1}$ 的取值时，按照下式进行计算：

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} P(1) \\ P(\omega) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

当给定多项式的取值($P(1), P(\omega), \dots, P(\omega^{n-1})$)=(v_0, v_1, \dots, v_{n-1})而要计算多项式的系数时，需要通过求解下列方程组得到 a_0, a_1, \dots, a_{n-1} ：

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix}$$

求解方程组通常非常耗时，一般情况下需要 $O(n^3)$ 时间

把等式写成 $\underline{V}(\omega) \underline{a} = \underline{v}$ ，此处 $V(\omega)$ 指左边的矩阵而
 $\underline{a} = (a_0, a_1, \dots, a_{n-1})$ 以及 $\underline{v} = (v_0, v_1, \dots, v_{n-1})$ 。只要 $V(\omega)$ 可逆，解 \underline{a}
可以写成 $\underline{a} = [V(\omega)]^{-1} \underline{v}$ 。事实上， $V(\omega)$ 总是可逆的，而且
它的逆矩阵有一个非常简单的形式

定理9.1: $[V(\omega)]^{-1} = \frac{1}{n} V\left(\frac{1}{\omega}\right)$

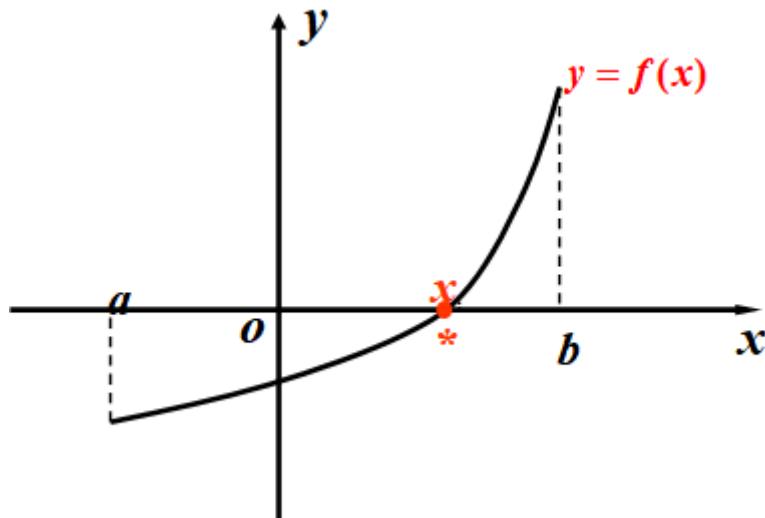
定理9.2: 如果 ω 是n次单位原根，那么 $1/\omega$ 也是n次单位原根。

所以只要将 ω 替换成 $1/\omega$ 就可以使用快速傅立叶变换的算法计算乘积 $V(1/\omega) \underline{v}$ ，该变换称为逆向傅立叶变换。

复杂性：总的来说，使用 $(n \log n)$ 次运算就可以计算两个多项式的乘积，注意到需要能够对复数进行加法和乘法运算。

非线性方程的数值解法

代数方程的求根问题是一个古老的数学问题。理论上，n次代数方程在复数域内一定有n个根(考虑重数)。早在16世纪就找到了三次、四次方程的求根公式，但直到19世纪才证明大于等于5次的一般代数方程不能用代数公式求解。因此需要研究数值方法求得满足一定精度要求的根的近似解。



Ch 10 归约

归约

假设有一个较复杂的问题P，而它看起来与一个已知的问题Q很相似，可以试着在两个问题间找到一个归约 (reduction, 或者transformation)。

归约是使用解决其它问题的“黑盒”来解决另一个问题。取决于问题解决的先后，归约可以达到两个目标：1) 如果已知Q的算法，那么就可以把使用了Q的黑盒的P的解决方法转化成一个P的算法；2) 如果P是一个已知的难题，或者特别地，如果知道P的下限，那么同样的下限也可能适用于Q。前一个归约是用于获取P的信息；而后者则是用于获取Q的相关信息。

归约让我们理解两个问题间的关系，它是一种可用于寻找解决某问题或其变形的技术

简单字符串匹配问题

CSM问题：设两个长度为n的字符串A=a₀a₁...a_{n-1}及B=b₀b₁...b_{n-1}，判断B是否A的循环平移。

方法1：修改6.7节中介绍的Knuth-Morris-Pratt算法来解决CSM

方法2：寻找特定的文本T及特定的模式P使得在T中寻找P等同于判定B是否A的一个循环平移

不同代表集

设 S_1, S_2, \dots, S_k 是一组集合。不同代表集 (System of Distinct Representative, SDR) 是一个集合 $R = \{r_1, r_2, \dots, r_k\}$, 其中 $r_i \in S_i$ 。由于 R 是集合, r_i 就必须是相异的, 即 R 刚好包含每个集合的一个代表元素。给定一组集合未必能找到一个 SDR。

问题: 给定一由有穷个有穷集合组成的组, 找出该组的 (任意一个) SDR, 或判断 SDR 是否存在。

设 $\text{card}(S)$ 是 S 的元素个数。

Hall 定理: 设 S_1, S_2, \dots, S_k 是一组集合, 其存在 SDR 当且仅当下面条件成立: $\text{card}(S_{i_1} \cup S_{i_2} \cup \dots \cup S_{i_m}) \geq m$, 对 $\{1, 2, \dots, k\}$ 的任一个子集 $\{i_1, i_2, \dots, i_m\}$, 即任意 m 个集合的并必须至少包含 m 个不同的元素, 对所有的 $1 \leq m \leq k$ 。

Hall 定理提出了很简单的条件, 但是很不幸, 必须检查所有可能的 2^k 个子集。

把它看成偶图匹配问题: 设 $G = (V, U, E)$ 是一个偶图, 使得对每个集合 S_i 均在 V 中存在一个顶点 v_i , 并且对所有可能的元素 (即对集合的并中的每个元素) 在 U 中都有一个顶点 u_j 。每个元素与包含它的所有集合相连, 则 SDR 就是 G 中规模为 k 的匹配, 可以应用 7.10 节中讨论的算法。

序列比较

序列比较问题: $A = a_1a_2\dots a_n$ 及 $B = b_1b_2\dots b_m$ 是两个字符串, 要逐字符地编辑 A 直到它变为 B 。允许插入、删除和替换三种编辑方式, 每个方式涉及一个字符, 且已知这些方式的代价, 目标是使编辑的代价最小。

6.8 的方法: 构造一个 $n \times m$ 的表, 其中第 i, j 项对应一个局部编辑, 包含了把 A 的前 i 个字符编辑成 B 的前 j 个字符的代价, 最终目标是完成表的右下角项 (第 n, m 项)。

另一种方法: 把表看成一个有向图。表中每一项对应图的一个顶点, 这样每个顶点对应一个局部编辑。当 v 对应的局部编辑能够经过一个以上编辑步骤到 w 对应的局部编辑时, 则存在一条边 (v, w) 。水平边对应插入, 垂直边对应删除, 而斜边对应替换。除了对应相同字符的斜边 (即无须替换) 的代价是 0, 其它每条边的代价是 1。

现在这个问题就变成了一个常规的单源最短路径问题了, 每条边都关联一个代价 (对应于编辑的代价), 而我们正在寻找从顶点 $[0, 0]$ 到顶点 $[n, m]$ 的最短路径。这样就已经把字符串编辑问题归约成单起点最短路径问题了。

一般来说寻找最短路径不比直接解决序列比较问题简单, 但是归约还是很有用的。

设想有下面各种变形的序列比较问题。编辑的代价不一定是以一个字符进行计算的, 在字符串正中插入一块字符的代价与在其它地方逐个插入相同数目的字符的代价可能是不一样的, 删除也是如此。换句话说, 要对一块字符的操作赋值, 无论其大小, 而不是对单步操作赋值。又如, 我们想为插入一个 k 字符的块赋予代价, 比方说 $l + ck$, 其中 l 是“起步” (start-up) 价, c 是后来的每个字符的代价。还有许多其它度量方法, 但都可以用最短路径形式对其进行建模, 而远比更改原来的问题来得容易, 可以在任何需要的地方添加边并对它们赋值, 而无须修改问题。

在无向图中寻找三角形

问题：设 $G=(V,E)$ 是一n个顶点m条边的无向图。设计一个算法判断G是否含有两两连接的三个顶点。

最直截了当的方法是查看所有三个顶点的子集，每个子集可以在常数时间内检查，因而时间复杂性是 $O(n^3)$ 。

设A是G的邻接矩阵。因为G是无向的，所以A对称。记 A^2 是矩阵A的平方，考虑 A^2 中各项与图G之间的关系。由矩阵乘法的定义， $A^2[i,j] > 0$ 当且仅当存在k使得 $A[i,k]$ 和 $A[k,j]$ 均为1，从图的角度看，当存在一个顶点k， $k \neq i$ 且 $k \neq j$ ，并且i和j都与k连接的时候 $A^2[i,j] > 0$ （假设图不包含自回路，这样对所有的i都有 $A[i,i]=0$ ）。这意味着存在一个关于i和j的三角形当且仅当i与j相连且 $A^2[i,j] > 0$ 。这样，在G中存在一个三角形当且仅当存在i和j使得 $A[i,j]=1$ 且 $A^2[i,j] > 0$ 。这样把在图中寻找三角形的问题归约为布尔矩阵相乘问题。⁹

线性规划

线性规划问题可形式化阐述如下：在满足不等式约束，等式约束和非负约束的条件下使目标函数最大化。

$$\begin{aligned} \min c(\bar{x}) &= \sum_{i=1}^n c_i x_i \\ s.t. \quad \bar{a}_i \cdot \bar{x} &\leq b_i \\ \bar{e}_j \cdot \bar{x} &= d_j \\ x_k &\geq 0 \end{aligned}$$

标准形式，松弛变量

网络流量问题归约到线性规划

$G=(V,E)$ 是一个有两个特殊顶点的有向图，一个是入度为0的s(源点)，一个是出度为0的t(汇点)。E中的每条边e赋予一个正的权值 $c(e)$ ，称为e的容量，用来度量经过这条边的最大流量，可为不存在的边赋予容量值0。这样的一个图称为网络。流量是一个作用在网络的边上的函数f，满足以下两个条件：

1. $0 \leq f(e) \leq c(e)$: 经过一条边的流量不能超过这条边的容量。
2. 对于所有的 $v \in V - \{s,t\}$ ， $\sum_u f(u,v) = \sum_w f(v,w)$: 进入一个顶点的流量之和等于离开这个顶点的流量之和(源点和汇点除外)。

设变量 x_1, x_2, \dots, x_n 表示所有边的流出量（这里n是边的数目）。目标函数是

$$c(\bar{x}) = \sum_{i \in S} x_i$$

其中S是从源出发的边的集合。对应容量约束的约束不等式是：

$$x_i \leq c_i \quad 1 \leq i \leq n$$

其中 c_i 是边*i*的容量。对应守恒约束的约束不等式是：

$$\sum_{\text{离开 } v_i \text{ 的 } x_i} x_i - \sum_{\text{进入 } v_i \text{ 的 } x_j} x_j = 0 \quad v \in V - \{s, t\}$$

最后，对所有的变量都应用非负约束

静态路由问题归约到线性规划

设 $G=(V,E)$ 是无向图，表示一个通讯网络。假设网络中每个节点有一个有限缓存，在一个单位时间内只能接收 B_i 条消息（设所有消息的长度均相同）。再假设任一链路可以传输的消息数目没有限制，每个节点均有无限的消息供应，问题是在单位时间内每条边应该传送多少条消息才能使在网络上的总消息数量最多。用图论表述，问题就是对边赋予权重，连接节点 V_i 的边的权重 $\leq B_i$ ，而使所有权重的和最大化。

这个图论问题可以很容易表述成一个线性规划问题，对每条边 e_i 关联一个变量 x_i ，表示经过 e_i 的消息数目。目标函数是，

$$c(\bar{x}) = \sum_i x_i$$

约束如下：

$$\sum_{\text{连接 } v_i \text{ 的 } e_i} x_i \leq B_i \quad x_i \geq 0 \quad 13$$

慈善家问题归约到线性规划

设有n个组织要向k个计算机系捐助，每个组织*i*在一年内都有总捐助限度 s_i ，对部门 j 的捐助同样也有限度 a_{ij} （对某些部门 a_{ij} 可能是0）。一般来说 s_i 比 $\sum_j a_{ij}$ 要小，因而每个组织都要做一些抉择。此外，设每部门也有收受总金额的限制 t_j （虽然这个约束不太现实，不过这是很有趣的）。目标是设计一个算法使总捐助金额最大（不管公平与否）

变量 x_{ij} 表示组织*i*对部门 j 的捐助数目。目标函数是

$$c(\bar{x}) = \sum_j x_{ij}$$

约束如下且所有的变量都是非负的。

$$x_{ij} \leq a_{ij} \quad \sum_{j=1}^k x_{ij} \leq s_i \quad \sum_{i=1}^n x_{ij} \leq t_j$$

分配问题归约到线性规划

对慈善家问题稍作更改：强调每个组织只能向一个部门捐助，而每个部门也仅接收一个组织的捐助。

对每条边(i,j)赋予一个变量 x_{ij} ，当选择该边时为1，反之为0。

目标函数变为：

$$c(\bar{x}) = \sum_{ij} a_{ij} x_{ij}$$

约束如下且所有的变量都非负：

$$\sum_{j=1}^k x_{ij} = 1 \quad \sum_{i=1}^n x_{ij} = 1$$

问题的界

在前面给出的最优算法的定义中，我们提到了问题的下界，如果任何一个求解问题P的算法是 $\Omega(f(n))$ 的，则该问题的下界是 $\Omega(f(n))$ 。实际上对于我们所有遇到的算法，都已经能够找到算法所需计算量的上界，但是找一个特定问题的下界却要困难得多。

有一类下界被称为平凡下界，可以在不借助任何计算模型或者进行复杂计算的情况下就能够得到。

对于一个没有排序的序列，为找出其中的最小数，必须检查序列中所有的数，因此在无序序列中进行查找的任何算法必须花费 $\Omega(n)$ 时间。

两个 $n \times n$ 矩阵相乘的任何算法都必须计算并输出恰好 n^2 个值，因此两个 $n \times n$ 矩阵相乘的任何算法的时间复杂性为 $\Omega(n^2)$ 。

要注意的是，平凡下界虽然不用借助任何计算模型或进行复杂的数学计算就能够推导出来，但是往往过小而失去意义。例如TSP问题的平凡下界是 $\Omega(n^2)$ ，因为对于n个城市的TSP问题，输入是 $n(n-1)/2$ 个距离，但是这个平凡下界是没有意义的，因为TSP问题至今还没有找到一个多项式时间算法。

决策树模型

平均情况下基于比较的排序算法的下界

定义：设L是二叉树T的叶节点集合，则T的外路径总长为

$$EPL(T) = \sum_{x \in L} \text{depth}(x), \text{ 其中 } \text{depth}(x) \text{ 表示节点 } x \text{ 的深度}$$

一棵二叉树T被称为有最小外路径总长的二叉树，如果其 $EPL(T)$ 是所有有同样叶节点个数的二叉树中最小的

一棵有最小EPL的二叉树必然是每个内节点恰好有两个子节点，并且所有叶节点必定出现在最底下两层

若具有最小EPL的二叉树T有x个叶节点，则 $EPL(T) > x(\lceil \log_2 x \rceil - 1)$

定理：若各种输入情况等概率出现，则任何一个基于比较的排序算法平均需要至少 $\log_2 n - 1$ 次比较，其中n为待排序的数字个数

若 T 在 $h-1$ 层的内部节点有 m

$$0 < m \leq 2^{h-1}$$

$h-1$ 层有 $2^{h-1} - m$ 个叶节点

h 层有 2^m 个 ...

叶节点总数 $\underline{2^{h-1} + m} \leq S$

$$EPL(\bar{T}) > (h-1)(2^{h-1} + m) = (h-1)$$

$$S \leq 2^h$$

$$2^{h-1} < S \leq 2^h$$
$$\therefore h-1 < \log S \leq h \quad \underline{h = \lceil \log S \rceil}$$

下界的归约

基本原理：如果解决问题Q的算法无需增加太多运行时间就可改为解决问题P的算法，那么问题P的下界也能适用于问题Q。

寻找简单多边形算法复杂度的下界

考虑用一个简单闭多边形连接平面上一系列点的问题，我们已经知道如何用排序来解该问题，在某些假设下，这个问题是不可能比排序来得更快。所以，不改进排序方法就无法改进简单闭多边形问题的求解速度（改进意味着速度的提高大于常数因子）。

定理10.1：给定一个在 $O(T)$ 时间内的简单多边形问题的（黑盒）求解算法，就可以在 $O(T+n)$ 时间内进行排序。

排序问题 $x_1 \dots x_n$

已知简单多边形算法 $O(T)$

1). $x_i \rightarrow \theta_i$ (角度) $O(n)$

2). 根据 θ_i 把 x_i 映射到单位圆上

\vdots \dots $O(n)$

3) 把这些点构成简单多边形

4) 按逆序输出 $\rightarrow x_1 \dots x_n$ 的序
 $O(n)$

推论10.2: 在决策树模型下, 在平面上寻找连接一系列给定点的简单多边形, 在最坏情况下需要 $\Omega(n \log n)$ 的比较。

关于矩阵的简单归约

问题: 特殊的矩阵相乘是否会更简单?

定理10.3: 如果有算法在 $O(T(n))$ 时间内计算两个 $n \times n$ 的对称实矩阵相乘, 满足 $T(2n)=O(T(n))$, 那么同样有算法在 $O(T(n)+n^2)$ 时间内计算两个 $n \times n$ 的任意实矩阵相乘。

定理10.4: 如果有算法在 $O(T(n))$ 时间内计算 $n \times n$ 实矩阵的平方, 满足 $T(2n)=O(T(n))$, 那么同样有算法在 $O(T(n)+n^2)$ 时间内计算两个 $n \times n$ 的任意实矩阵相乘。

$O(n^3)$

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

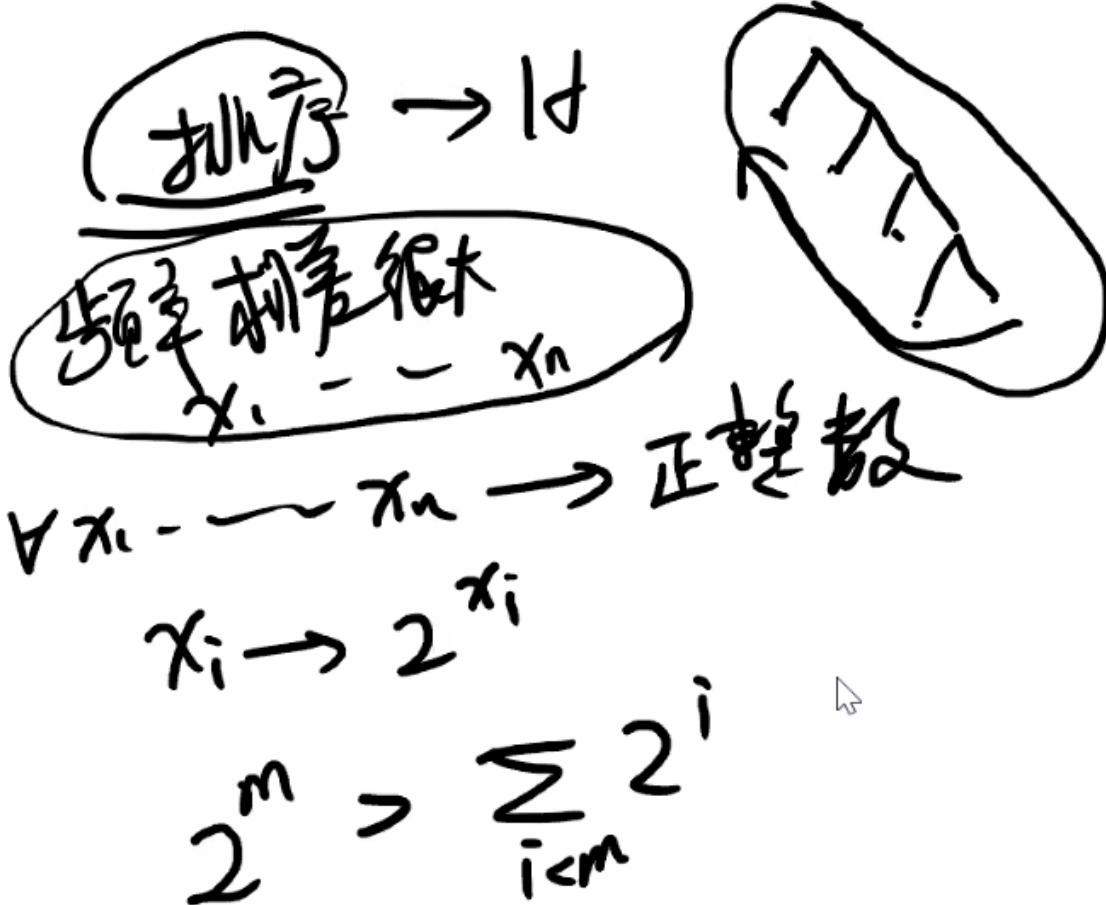
$\forall A_{n \times n} B_{n \times n}$

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}$$

$$O(n^2) \quad \cancel{O(n^2)} \quad O(n^2)$$

$$A \ B \quad \underbrace{\begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}^2}_{=} = \boxed{\begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}}$$

Huffman $\approx n \log n$



常见的错误

最普遍的问题是以错误的方式使用归约

归约过程中不应该加入明显低效的操作。

Ch 11 计算机不是万能的

我们的问题分类法

目前计算机已用来求解各类问题，但是其能力不是没有极限的，因此计算机并不能用来求解所有的问题，“什么能计算”是计算机科学的根本问题。

首先问题可以分为计算机能够求解和不能求解，这一划分基于问题是否能够在有限时间内解决，与计算时间长短无关。

在计算机能够求解的问题中，以是否能够设计出多项式时间算法为分界线将其分为易解的和难解的两类，对于难解的问题，虽然可以求解，可以为其设计算法，但由于算法的时间复杂性为指数型，不具有实用价值，并且这些问题目前还设计不出多项式时间的算法，将来也不大可能发现多项式时间的算法。

不可计算的问题

场景：计算机正在执行一个程序，我们坐在边上等待程序结束。

问题：当过了很久程序还没结束时，我们该怎么办呢？

1) 推测程序中可能出现了无穷循环，永远不会结束，这样我们就必须强行中断程序运行甚至重启计算机。

2) 也许是因为计算太复杂导致时间过长呢？这样的话，我们就该继续等待。

如何选择？

设想：要是有这么一个程序P，其功能是以另一个程序Q的代码作为输入，并分析Q中是否包含无穷循环。

很遗憾，这样的程序P是不存在的！

这就是所谓停机问题（Halting problem）。

停机问题

从算法设计角度看，停机问题就是要设计算法halt，它的输入为另一个程序prog的源代码，输出为prog是否无穷循环。由于prog的行为不仅依赖于它的源代码，还依赖于它的输入数据，因此为了分析prog的终止性，还要将prog的输入数据data交给halt。由此可得halt的定义说明：

算法：停机判别算法halt

输入：程序prog的源代码，以及prog的输入数据data，都以字符串形式表示

输出：如果prog在data上的执行能终止，则输出True，否则输出False

在停机问题中，正常情况下是想运行输入为data的程序prog，即prog(data)，但又不知道这个执行过程能不能终止，于是希望将prog的代码和data交给停机分析算法halt，由halt来判断prog(data)的终止性。

假如已经设计出了停机分析算法halt，虽然不知道halt的实现过程，但基于halt算法设计如下算法strange

```
算法 strange
输入：字符串p
1. result = halt(p,p)
2. if result == True      {halt算法判定p(p)终止}
3.   while True
4.     pass
5.   end while
6. else                  {halt算法判定p(p)不终止}
7.   return
8. end if
```

那么运行strange(strange)的结果是什么呢？

strange首先调用halt(p,p)，这里的关键是传递给halt的两个输入都是p，亦即要分析程序p以它自己为输入数据时——即p(p)——运行是否终止。strange根据halt(p,p)的分析结果来决定自己接下去怎么做：如果结果为True，即p(p)能终止，则strange进入一个无穷循环；如果结果为False，即p(p)不终止，则strange就结束。

strange程序看上去有点费解，但只要halt存在，strange在编程方面显然没有任何问题。

将strange自身的源代码输入给strange时会发生什么？更确切地，strange(strange)能否终止？

参照上面的strange代码来分析。假如调用strange(strange)不终止，那必然是因为执行到了代码中条件语句的if result == True部分，即halt(strange,strange)返回了True，这又意味着strange以strange为输入时运行能终止！另一方面，假如调用strange(strange)能终止，那必然是因为执行到了条件语句的else部分，即halt(strange,strange)返回了False，这又意味着strange以strange为输入时运行不能终止！总之，我们得到了如下结论：

若strange(strange)不终止，则strange(strange)终止；

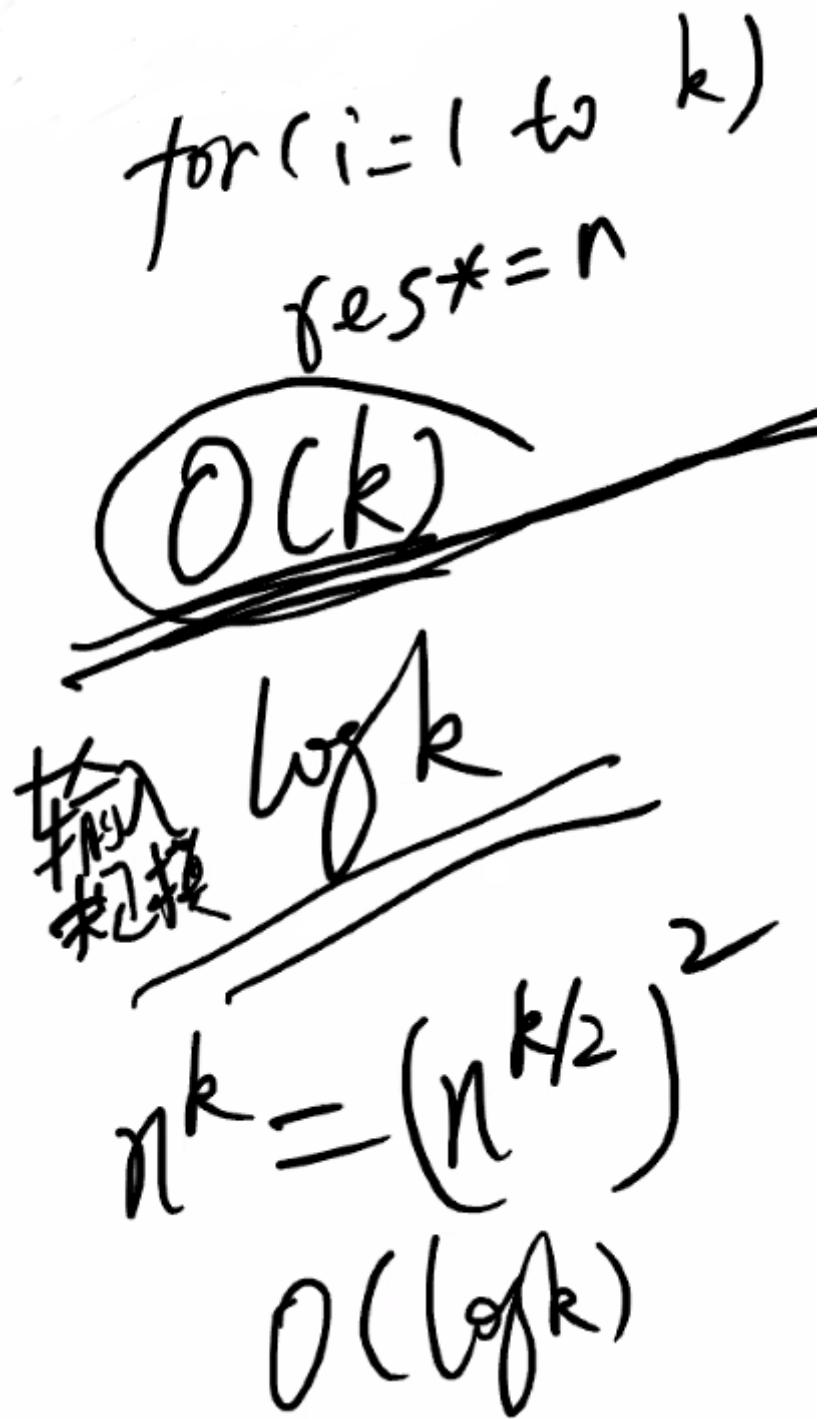
若strange(strange)终止，则strange(strange)不终止。

这样的结论在逻辑上显然是荒谬的。导致这个矛盾的原因在于假设了halt的存在，并利用了halt的功能
停机问题是一个不可解问题，即没有一个计算机程序能够确定另外一个计算机程序是否会对一个特定的输入停机。

一般的观点

运行时间与输入规模的某个多项式函数相关，一般称这样的算法是高效的(efficient)，相应的问题是易处理的(tractable)。换句话说，如果算法的运行时间是 $O(P(n))$ ，这里 $P(n)$ 是输入规模 n 的多项式函数，那么可称该算法是高效的。用P（指多项式时间）表示所有可以被高效算法所解决的**问题类**。 O 带log的也是P类问题，快排是算法不是P类问题。

求 n^k



分治法是P类问题

NP完全问题不存在高效算法

判定问题

判定问题 (decision problem)，即只考虑那些答案为肯定或者否定的问题。

大多数问题都可以转化成判定问题。

判定问题可以视为语言识别问题，假设 U 是由该判定问题的各种可能的输入所组成的集合，而 $L \subseteq U$ 是回答为肯定的全部输入所组成的集合，一般称 L 为与该问题相关的语言。

多项式归约

设 L_1 和 L_2 分别是输入空间 U_1 和 U_2 的两个语言，如果存在多项式时间算法可以将每个输入 $u_1 \in U_1$ 转化成另一个输入 $u_2 \in U_2$ ，使得 $u_1 \in L_1$ 当且仅当 $u_2 \in L_2$ ，则称 L_1 可以多项式归约到 L_2 。该算法是输入 u_1 的大小的多项式函数，同时 u_2 的大小是 u_1 的大小的多项式函数。如果有 L_2 的算法，就可将这两个算法组合起来生成 L_1 的算法。

定理 11.1： 如果 L_1 可以多项式归约到 L_2 并且 L_2 有一个多项式时间的算法，那么 L_1 也存在一个多项式时间的算法。

归约的概念并不是对称的

如果两个语言 L_1 和 L_2 可以相互多项式归约到对方，则称它们是多项式等价的，或等价的。

定理 11.2： 如果 L_1 可以多项式归约到 L_2 并且 L_2 可以多项式归约到 L_3 ，那么 L_1 可以多项式归约到 L_3 。

$L_1 \not\propto_{\text{poly}} L_2$

AC: 转化算法

AL_2 : L_2 的算法

$\forall u_1 \in U_1 \xrightarrow{L} u_2 \in U_2$

$u_2 \xrightarrow{\text{AL}_2} \{T, F\} \rightarrow \begin{cases} u_1 \\ \text{否} \end{cases}$

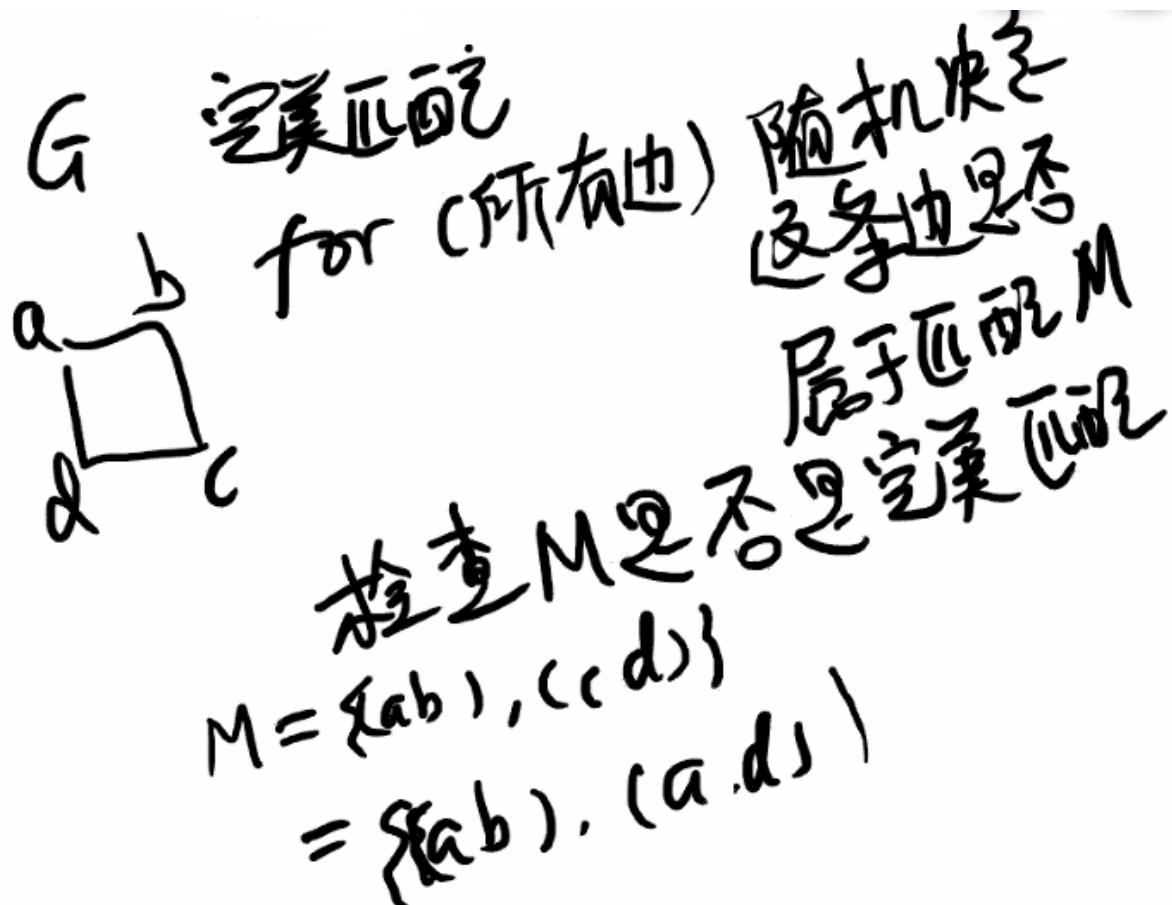


非确定算法

除具备确定性算法的所有常规操作外，非确定算法有一个非常强的基本操作，称为nd-choice。该基本操作与固定数目的一些可选项结合在一起使用，每次进行选择，从而算法可沿着不同的路径进行。给定输入 x ，非确定算法交替执行常规的确定性步骤和nd-choice操作，最终决定是否接受 x 。

确定性和非确定性算法最突出的差异：识别语言的方式。

一个非确定性算法能识别语言 L 是指给定输入 x , $x \in L$ 当且仅当可以将算法执行过程中的每一次nd-选择转变成确定的选择，使得算法能接受 x 。即存在一种True的输出情况。不能接受 x 即所有的输出情况都是False



对于输入 $x \in L$, 算法的运行时间是指到达接受状态的最短执行序列长度, 非确定算法的运行时间是指对于所有 $x \in L$ 在最坏情况下的运行时间



NP

存在运行时间关于输入是多项式函数的非确定算法的问题所组成的类称为NP类

决定P和NP之间关系的问题称为P=NP问题。

一般随机生成候选解，然后检查解是否满足要求

TSP问题是NP问题。输入 G, k 检测随机路径是否小于k

求解是困难的，检验是简单的

$P \subseteq NP$ NP中恰好不涉及非确定算法

定义：如果所有NP中的问题都可以多项式归约到问题X，则称X为NP难（NP-hard）问题。

定义：如果问题X满足（1）X在NP中，（2）X是NP难的，那么称X为NP完全问题。

引理11.3：如果问题X满足（1）X属于NP，（2）对于某个NP完全问题Y，Y可以多项式归约到X，那么X是NP完全问题。

①. X不会比其它NP问题容易
②. 如果有一个NPH问题属于P
则 $P = NP$



排序是否是NPC

$P = NP$
如 $S \in NPC$. 所有NP问题都可解决?
如 $S \notin NPC$ $P \neq NP$

可满足性问题——第一个NPC问题

设 S 是一个用合取范式 (CNF, Conjunctive Normal Form) 表示的布尔表达式，也就是说 S 是一些和 (或) 的积 (与)，如果存在使整个表达式取值为1的布尔变量赋值，则称该布尔表达式是可满足的。

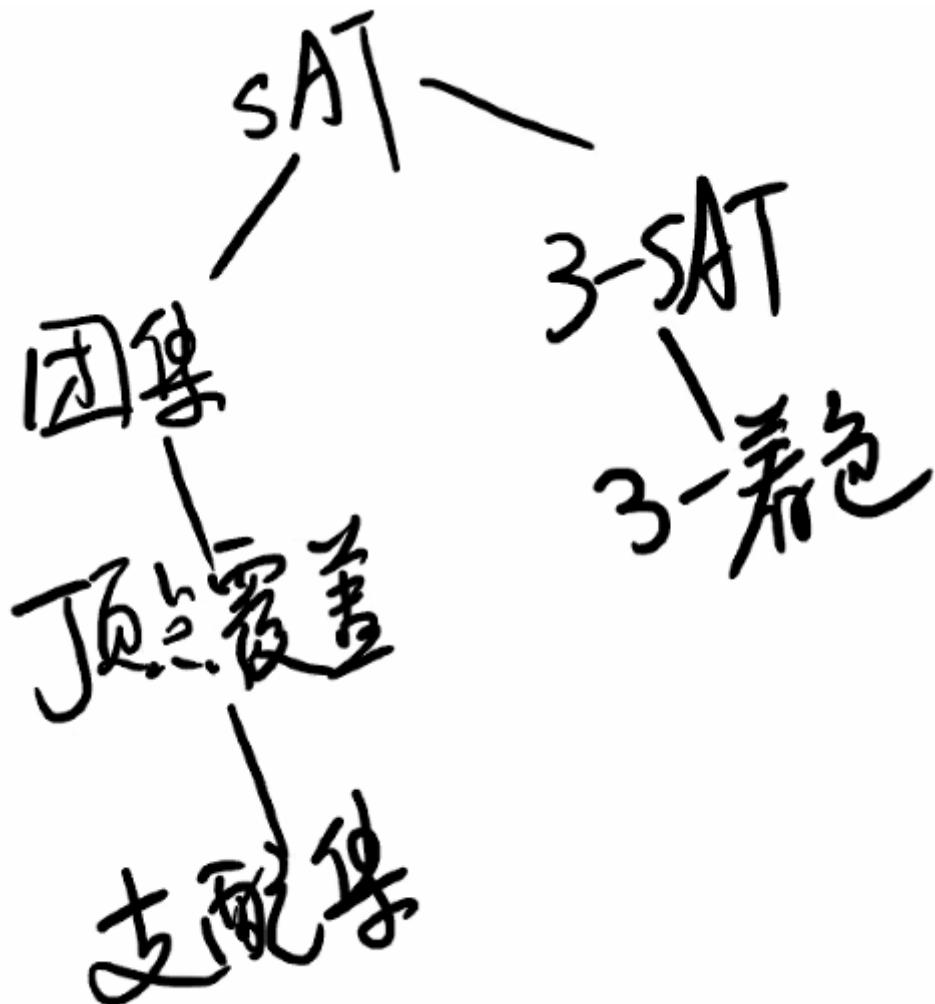
SAT问题：判定所给定的表达式是否可满足（不必找到使之满足的赋值）。

可以猜测一个真值指派并且在多项式时间内验证它是否满足表达式，故SAT问题在NP中。

Cook定理：SAT问题是NP完全的。

NP完全性的证明

要证明一个新问题是NP完全的，首先要证明其属于NP，这通常是（但并非总是！）容易的，接着要将一个已知的NP完全问题在多项式时间内归约到这个新问题。



顶点覆盖 VC问题

设 $G=(V,E)$ 是一个无向图， G 的顶点覆盖是一个顶点集合，满足 G 中所有的边都至少和该集合中的一个顶点相关联。

问题：给定无向图 $G=(V,E)$ 和一个整数 k ，判定 G 是否有包含 $\leq k$ 个顶点的顶点覆盖。

最优化形式：给定一个无向图，求出最小顶点覆盖

定理11.4：顶点覆盖问题是NP完全的。

将团问题归约到顶点覆盖问题

∀ 团集问题 $G = (V, E)$ k

令 $\bar{G} = G(V, \bar{E})$ 为 G 的补图

得到 VC 问题 $(\bar{G}, n-k)$

设 $C = (U, F)$ 为 G 的团集.

$$|U| \geq k$$

$V-U$ 是 \bar{G} 的 VC, ~~$|V-U| \leq n-k$~~

$\forall \underline{(u, v)} \in \bar{E}$

$\because (u, v) \notin E$

u 和 v 不可能同时属于 U

u 和 v 至少有一个属于 $V-U$

顶点覆盖有解 \rightarrow 团集有解

若 D 是 \bar{G} 的 VC 且 $|D| \leq n-k$

$\forall (u, v) \in \bar{E}, u, v$ 至少有一个属于 D

$\therefore \bar{G}$ 中没有边由 $V-D$ 的边相连构成

$\therefore V-D$ 就可以构成 G 的团集

$\forall u, v \in V-D \quad (u, v) \in \bar{E}$

支配集问题

设 $G=(V, E)$ 是一个无向图，如果顶点集合 D 满足， G 中的所有顶点要么在 D 中要么与 D 中至少一个顶点相邻，则称 D 为支配集。

问题：给定无向图 $G=(V, E)$ 和整数 k，判定 G 中是否有一个包含 $\leq k$ 个顶点的支配集。

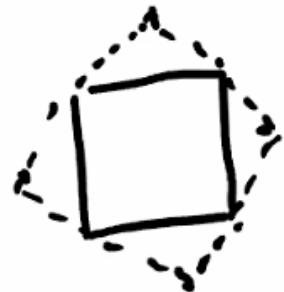
定理 11.5：支配集问题是 NP 完全的。

将顶点覆盖问题归约到支配集问题

$\text{VC}(G, k) \leqslant \text{支配集问题}$
 (V, E)

对于边 (v, w) ，增加上 u，反边 (v, u)
 (u, w)

得到 $G' = (V', E')$



增加 $|E|$ 个
2|E| 条边

支配集问题 (G', k)

设 $D \not\supseteq G'$ 的支配集

如果 D 中有新增顶点 u ,
用 v 或 w 替换 u , D 仍是支配集

可以假设 D 中仅包含 G 中的顶点

$\because D$ 支配所有新顶点
 $\therefore D$ 一定包含图中每条边的一个端点

$\therefore D \supseteq G$ 的 VC

若 $C \not\supseteq G$ 的 VC

由于所有边被 C 覆盖
 \therefore 所有新顶点都全被 C 所支配

$\therefore C$ 是 G' 的支配集

3SAT问题

3SAT问题是一般SAT问题的简化，3SAT的实例是指每个子句中恰好含有3个变量的布尔表达式。2SAT是P类问题

问题：给定以CNF形式出现并且每个子句恰好含有3个变量的布尔表达式，判定其是否可满足。

定理11.6：3SAT问题是NP完全的。

一般SAT问题归约到3SAT问题

$$\begin{aligned}
 C &= x_1 + x_2 + \dots + x_k \\
 k \geq 4 &\quad \exists i \text{ 使 } k-i \text{ 个变量} \quad C = \underline{x+y+\bar{z}} \\
 C' &= \underline{(x_1+x_2+y_1)} (x_3+\bar{y}_1+y_2) (x_4+\bar{y}_2+y_3) \dots (x_i+\bar{y}_{i-2}+y_{i-1}) \dots \\
 &\quad \dots (x_{k-3}+\bar{y}_{k-5}+y_{k-4}) (x_{k-2}+\bar{y}_{k-4}+y_{k-3}) \underline{(x_{k-1}+x_k+\bar{y}_{k-3})} \\
 C \text{ 可满足} &\Leftrightarrow C' \text{ 可满足} \\
 C \text{ 可满足} &\quad \therefore \exists i \quad x_i = 1 \quad \text{全 } y_1 \dots y_{i-2} = 1 \quad \text{其余为0} \\
 C \text{ 可满足} &\quad \therefore \exists i \quad x_i = 1 \quad \text{全 } y_1 \dots y_{i-2} = 1 \quad \text{其余为0} \quad C' = 1 \\
 C \text{ 可满足} \text{ 反证} &\quad C = 0 \quad \text{所有 } x_i = 0 \quad C' = \frac{y_1}{1} \cdot (\bar{y}_1+y_2) \cdot \frac{y_2}{0} \cdot (\bar{y}_2+y_3) \cdots \frac{y_{k-4}}{0} \cdot (\bar{y}_{k-4}+y_{k-3}) \cdot \frac{y_{k-3}}{0} \\
 C' &= 0 \text{ 矛盾} \\
 C &= x + x_2 \\
 C &= x \quad C' = (x+y+\bar{z}) (x+\bar{y}+\bar{z}) (x+y+\bar{z}) (x+\bar{y}+\bar{z})
 \end{aligned}$$

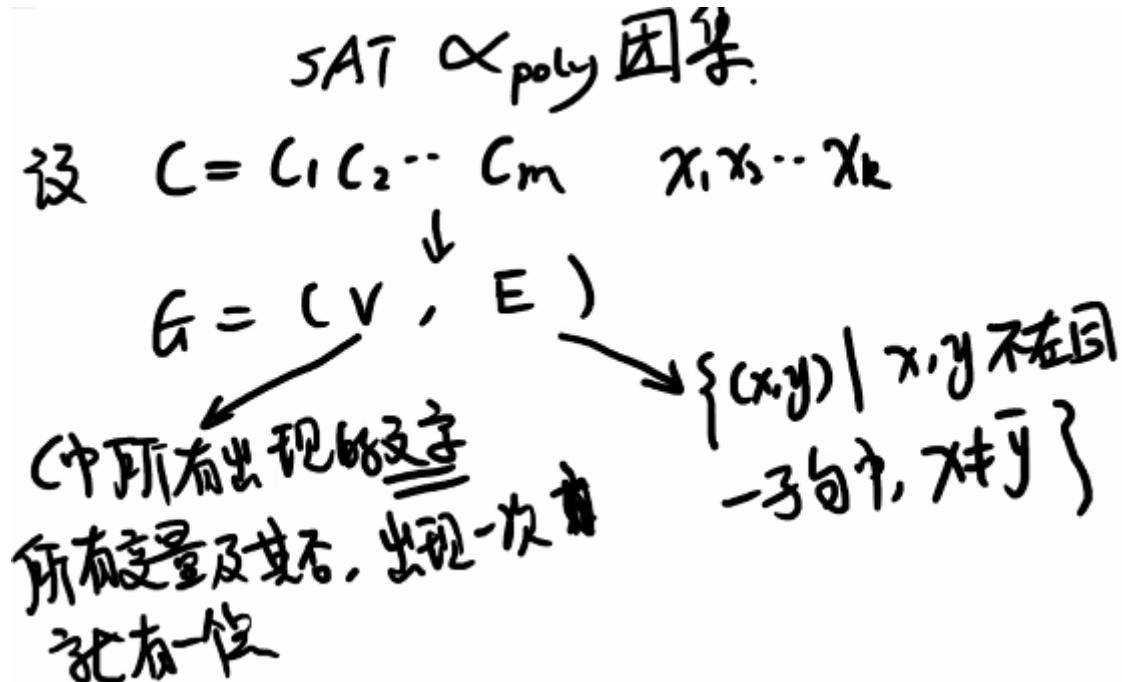
团问题

给定无向图 $G=(V,E)$, G 中的一个团 C 是 G 的一个子图, 满足 C 中的任何两个顶点均相邻, 换句话说, 团即完全子图。

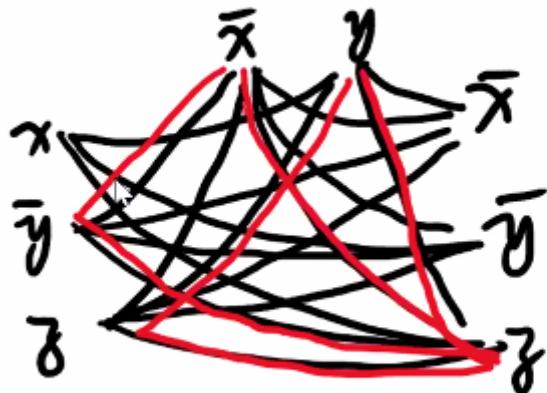
问题: 给定无向图 $G=(V,E)$ 和整数 k , 判定 G 是否包含一个大小 $\geq k$ 的团。

定理11.7: 团问题是NP完全的。

将SAT问题归约到团问题



$$C = (x + \bar{y} + z)(\bar{x} + y)(\bar{x} + \bar{y} + \bar{z})$$



$$k=m$$

规模为m的团集 = m个不同子句中对m个文字赋值为1
 文字x和y间有边意味着x和y可以同时取真
 (可满足) \Leftrightarrow 存在一个对于m个子句 m个文字的真值指派 \Leftrightarrow G中存在一个规模为m的团集



3着色问题

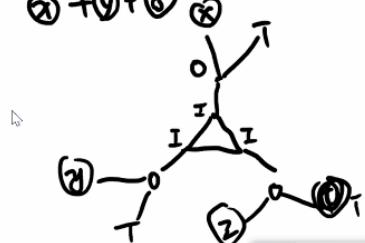
设 $G=(V,E)$ 是无向图， G 的有效着色是指对所有顶点的颜色指派，使得每个顶点被指派一种颜色并且相邻顶点不被指派成相同颜色。

问题：给定无向图 $G=(V,E)$ ，判定 G 是否可以被3种颜色着色。

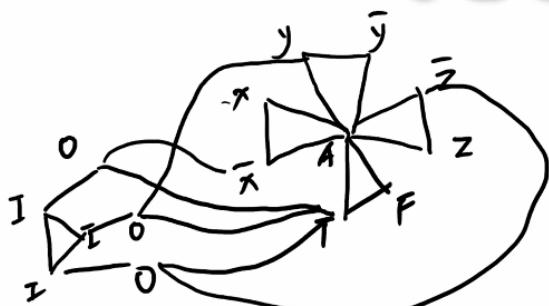
定理11.8：3着色问题是NP完全的。

将3SAT问题归约到3着色问题

1. \leftarrow 构造 ΔM , 三个结点 T, F, A^*
- 2) 对每个变量 x 构造 ΔM_x
这些 Δ 与 M 共享顶点 A^*
- 3) 对任一子句 3 插入 6 个 Δ 并将其连到现有顶点。
 $\textcircled{2} + \textcircled{3} + \textcircled{4}$



$$E = (\bar{x} + y + z)(\bar{x} + \bar{y} + z)(\bar{x} + y + \bar{z})$$



④ ⑤ ⑥ 三个至少有一个为T
∴ ④, ⑤, ⑥ 都与A相连
∴ 不能为A
~~如④ ⑤ ⑥ 都为F ∵ 三个O都必须加A~~ X

C 可满足 $\Leftrightarrow G$ 3着色

C 可满足, 令此时的真值指派为 G 着色
每个子句中至少一个文字为真, 则相应的顶点 O 着色 F
其余一一为 A
再对相应的顶点 O 着色

G 可 3 着色, 由 M 中的顶点着色, 对应真值指派
由图的构造可知, 每个子句中至少一个文字为 T

一般的经验

首先, 证明 Q 属于 NP, 这通常是 (但非总是) 简单的。接着要选择一些看上去与 Q 相关或者类似的已知的 NP 完全问题, 但有时问题看上去差异很大, 因此很难选择“类似”的目标, 这只能通过经验来把握, 一般的方法是尝试对几个问题归约, 直到发现一个成功的例子为止。

归约是从一个已知 NP 完全问题的任意实例到问题 Q, 通常最容易犯的错误是反方向进行归约

在归约过程中可以有一定的自由度: Q 包含的参数可以将它设置成任何固定的值; 只要归约可以在多项式时间完成, 其效率并不重要。不仅可以忽略常数因子 (例如, 使问题规模增加一倍), 还可以使问题规模扩大平方倍!

通用的技术: 证明一个 NP 完全问题是 Q 的特例是最简单的; 局部归约

其它常见 NP 完全问题

哈密尔顿问题: 给出无向图 $G = (V, E)$, 是否存在一条遍历每个顶点一次且仅一次的路径?

旅行商问题: 给出加权无向图 $G = (V, E)$, 求出遍历每个顶点一次且仅一次的最短路径

0-1 背包问题: $U = \{u_1, u_2, \dots, u_n\}$ 是一个准备放入容量为 C 的背包中的 n 个物品的集合, 第 i 个物品 u_i 具有体积 s_i 和价值 v_i , 要求从这 n 个物品中挑选出一部分装入背包, 在不超过背包容量的前提下使背包中物品的价值最大。

装箱问题: 给出大小为 s_1, s_2, \dots, s_n 的 n 个物品, 能否最多用 k 个容量为 C 的箱子将这些物品装入?

集合覆盖问题: 给定集合 X 以及 X 的子集族 F, 是否存在 F 中的 k 个子集, 它们的并集是 X?

作业调度问题: 有 n 个作业 J_1, J_2, \dots, J_n , 每项作业 J_i 的需要的机器运行时间为 t_i , 那么能否在时间 T 内利用 m 台机器完成所有 n 项作业?

处理NP完全问题的技术

NP完全问题理论能识别那些不太可能存在多项式时间算法的问题，但如何解决？

识别出其中能够多项式时间求解的特例，如2SAT问题就属于P类问题；

回溯法与分支限界法：这两种算法在求解问题时，虽然并不是多项式时间算法，但在解决其中不少实例时仍然会取得较快的速度；

确保一定性能的多项式时间算法：虽然其解与最优解之间存在一定差异，但却能够在多项式时间内完成，这类算法称为近似算法

随机算法：虽然不能够保证得到正确的解，但能够在多项式时间内完成。

回溯法

回溯法是一种有组织的穷举搜索方法，但经常能够在不试探所有可能性的情况下得到解，适用问题的特性是虽然可能有很多潜在的解，但真正需要考虑的并不多。

3着色问题

潜在的解的数目为 3^n ，即对n个顶点进行3着色的各种可能。

除非图中没有边，有效解的数目将远小于 3^n ，这是因为边对着色加了限制。为了探求所有顶点的着色方案，可首先对其中一个顶点任意着色，并且在满足由边所带来的约束条件（即相邻顶点必须着色不同）的同时，继续给其他顶点着色。当对一个顶点进行着色时，可尝试各种与先前顶点的着色不冲突的方案。该过程可以通过树遍历算法来完成

```
Algorithm 3-coloring(G,var U)
Input: G=(V,E) (an undirected graph), and U (a set of vertices that have already
       been colored together with their colors) {U is initially empty}
Output: An assignment of one of three colors to each vertex of G
begin
    if U=V then print "coloring is completed"; halt
    else
        pick a vertex v not in U;
        for C:=1 to 3 do
            if no neighbor of v is colored with color C then
                add v to U with color C;
                3-coloring(G,U)
end
```

分枝限界法

分枝限界法是回溯法求解涉及寻找某个目标函数最小（或最大）值问题的一种变形方法

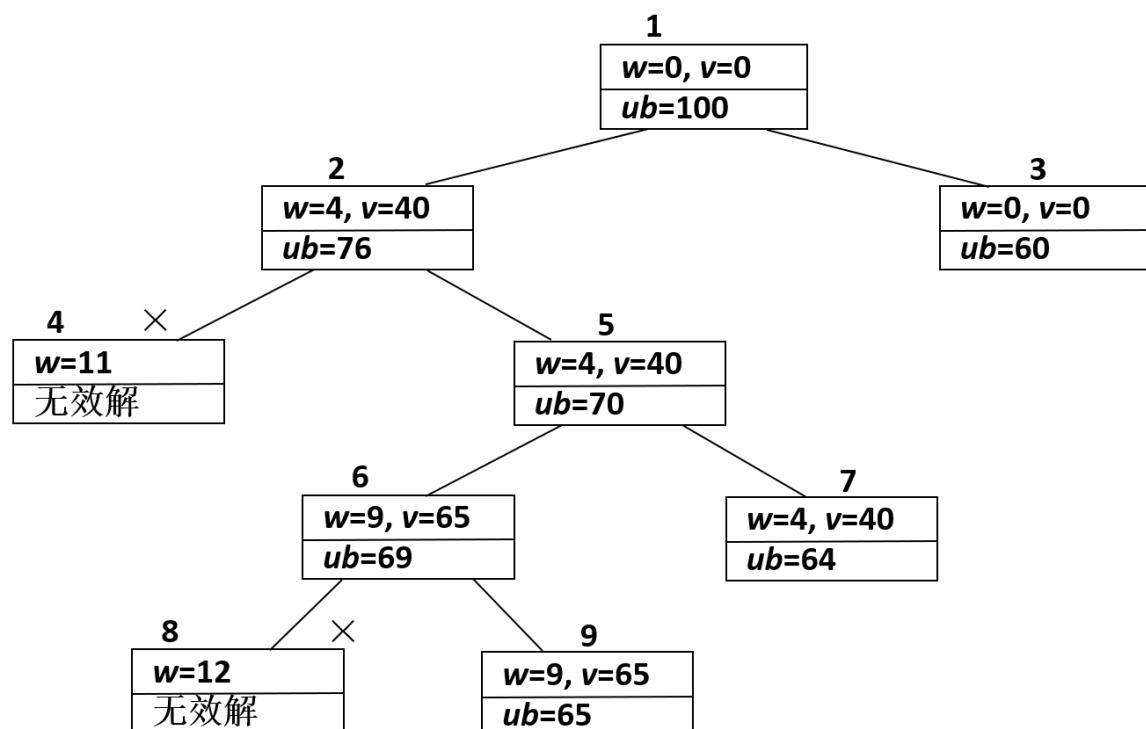
0/1背包问题

假设有4个物品，其重量分别为(4, 7, 5, 3)，价值分别为(40, 42, 25, 12)，背包容量 $W=10$ 。首先，将给定物品按单位重量价值从大到小排序，结果如下：

物品	重量(w)	价值(v)	价值/重量(v/w)
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

- 求得近似解为(1, 0, 0, 0)，获得的价值为40，这可以作为0/1背包问题的下界。
- 考虑最好情况，背包中装入的全部是第1个物品且可以将背包装满，则可以得到一个非常简单的上界的计算方法： $ub=W \times (v_1/w_1)=10 \times 10=100$ 。
- 目标函数的界：[40, 100]。
- 限界函数为：

$$ub = v + (W - w) \times (v_{i+1}/w_{i+1})$$



搜索过程：

1. 在根结点1，没有将任何物品装入背包，因此，背包的重量和获得的价值均为0，根据限界函数计算结点1的目标函数值为 $10 \times 10 = 100$ ；
2. 在结点2，将物品1装入背包，因此，背包的重量为4，获得的价值为40，目标函数值为 $40 + (10 - 4) \times 6 = 76$ ，将结点2加入待处理结点表PT中；在结点3，没有将物品1装入背包，因此，背包的重量和获得的价值仍为0，目标函数值为 $10 \times 6 = 60$ ，将结点3加入表PT中；
3. 在表PT中选取目标函数值取得极大的结点2优先进行搜索；
4. 在结点4，将物品2装入背包，因此，背包的重量为11，不满足约束条件，将结点4丢弃；在结点5，没有将物品2装入背包，因此，背包的重量和获得的价值与结点2相同，目标函数值为 $40 + (10 - 4) \times 5 = 70$ ，将结点5加入表PT中；
5. 在表PT中选取目标函数值取得极大的结点5优先进行搜索；
6. 在结点6，将物品3装入背包，因此，背包的重量为9，获得的价值为65，目标函数值为 $65 + (10 - 9) \times 4 = 69$ ，将结点6加入表PT中；在结点7，没有将物品3装入背包，因此，背包的重量和获得的价值与结点5相同，目标函数值为 $40 + (10 - 4) \times 4 = 64$ ，将结点6加入表PT中；
7. 在表PT中选取目标函数值取得极大的结点6优先进行搜索；
8. 在结点8，将物品4装入背包，因此，背包的重量为12，不满足约束条件，将结点8丢弃；在结点9，没有将物品4装入背包，因此，背包的重量和获得的价值与结点6相同，目标函数值为65；
9. 由于结点9是叶子结点，同时结点9的目标函数值是表PT中的极大值，所以，结点9对应的解即是问题的最优解，搜索结束。

确保性能的近似算法

近似算法的解和最优解的差距并不大

计算时间为多项式时间

顶点覆盖问题

近似算法：假设 $G=(V,E)$ 是一个图， M 是 G 的一个极大匹配，由于 M 是一个匹配，其所包含的边没有共同顶点，又因为 M 是极大的，所有其他边至少和 M 中的一条边有相同顶点。

定理11.9：与极大匹配 M 中的边相关联的所有顶点构成一个顶点覆盖，并且其大小不超过最小顶点覆盖大小的2倍。

寻找极大匹配的方法是简单收集边直到所有边都被覆盖到。但寻找最小极大匹配（即有最少边个数的极大匹配）问题也是NP完全的

M是极大匹配：
M的顶点集是VC
对于一个顶点覆盖，要覆盖所有边
就会覆盖M中的边
由于M是匹配，M中的边不能关联
M中的两条边

M中至少有一般的点属于VC

一维箱柜包装问题

箱柜包装问题是指将不同大小物体打包到规定大小的箱柜中，并且箱柜使用的数目要尽可能少。

问题：设 x_1, x_2, \dots, x_n 是介于0和1之间的实数，将这些数划分成尽可能少的子集使得每个子集中的数总和不超过1。

First Fit算法：将 x_1 放入第一个箱柜，接着对于每个 i ，将 x_i 放入能容纳它的第一个箱柜，或者当使用过的箱柜都不能容纳它时则使用一个新的箱柜。

定理11.10：First Fit算法至多需要 $2OPT$ 个箱柜(常数2还可以减为1.7)。

Decreasing First Fit算法：先逆序排序，然后再使用First Fit。

定理11.11：Decreasing First Fit算法至多需要 $11/9OPT + 4$ 个箱柜。

欧几里德旅行商问题

问题： C_1, C_2, \dots, C_n 是平面中的点集合，其对应于n个城市的位置；试找到一条长度最短的哈密尔顿回路

首先计算最小代价生成树（这里代价=距离），树的代价不会超过最佳TSP长度。

考察由深度优先搜索遍历树（从任意顶点出发）所形成的环路，每条边恰好被遍历了两次，所以这个环路的代价是这棵最小生成树代价的两倍，因而也不会超过最短TSP行程的两倍。可以通过选取直接路线而非回溯的方法将这个环路转变成一个TSP路径，即直接到第一个新顶点而不是回溯。由于采用欧几里德距离，这样形成的TSP路径长度不会超过最短TSP路径长度的两倍。

复杂度：运行时间主要由最小代价生成树算法的运行时间所决定的，对于欧几里德图为 $O(n \log n)$

改进

关键：高效地将树转变成欧拉图

欧拉图中所有节点的度均为偶数，考虑树中的所有度为奇数的节点，它们的个数一定是偶数（否则所有节点度数的总和将会是奇数，由于节点度数总和恰好是边数目的2倍，因而这是不可能的）。

对这棵树添加足够多的边，使得所有节点的度均为偶数，这样就得到了一个欧拉图。由于TSP路径将由这个欧拉环路（会走一些捷径）构成，因此要将额外添加的边的长度最小化

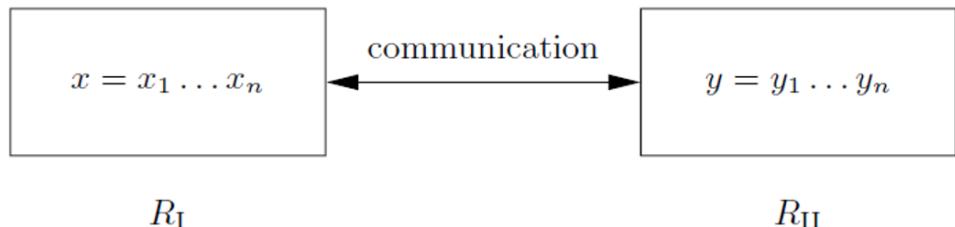
问题：给定平面中的一棵树，希望添加一些边使之成为一个欧拉图，并且要使添加边的总长最小。

对于每个度为奇数的顶点至少要添加一条边，尝试就添加一条。假定有 $2k$ 个度为奇数的顶点，如果添加 k 条边，每条边连接两个度为奇数的顶点，那么所有边的度都为偶数了。这样该问题就变成了匹配问题了，并且要找到一个最短长度匹配，使其覆盖所有度为奇数的顶点。对于一般图可以在 $O(n^3)$ 时间内找到权值最小的完美匹配，最近提出了一个针对欧几里德距离的算法，运行时间为 $O(n^{2.5} (\log n)^4)$ 。最终的TSP路径可以通过走捷径的方法从欧拉图获得

定理11.12：改进后算法所生成的TSP行程，其长度至多为最短TSP行程长度的1.5倍。

随机算法

考虑如下场景：



需要判断 R_I 和 R_{II} 的数据是否相同
需要多少通信量？

显然任何一种确定性算法需要在 R_I and R_{II} 之间传输至少 n 比特
若 $n = 10^{16}$ ，则要将这些数据全部正确地传送并不是一件容易的小事

算法

设 $x = x_1 x_2 \dots x_n$, $x_i \in \{0, 1\}$, 记

$$\text{Number}(x) = \sum_{i=1}^n 2^{n-i} \cdot x_i$$

初始状况： R_I 有 n 位序列 $x = x_1 x_2 \dots x_n$, R_{II} 有 n 位序列 $y = y_1 y_2 \dots y_n$.

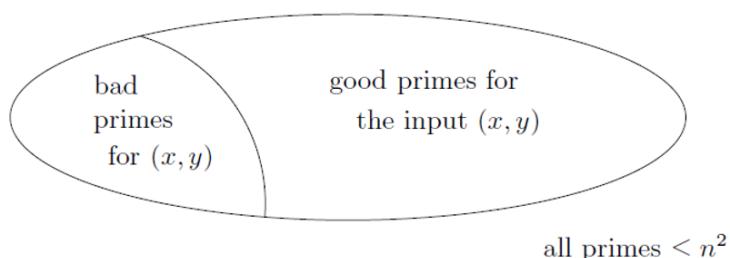
- 1: R_I 在 $[2, n^2]$ 内随机选择素数 p
- 2: R_I 计算得到整数 $s = \text{Number}(x) \bmod p$, 将 s 和 p 的二进制表示发送给 R_{II}
- 3: 收到 s 和 p 之后, R_{II} 计算 $q = \text{Number}(y) \bmod p$.
若 $q = s$, R_{II} 输出 “ $x = y$ ”
若 $q \neq s$, R_{II} 输出 “ $x \neq y$ ”

性能

所需通信量仅为 $4 \log n$

但这种算法可能出错

对任意的 (x, y) , 相应的坏素数的个数最多 $n-1$, 因此对于输入 (x, y) ($x \neq y$), 错误率最多为 $(\ln n^2)/n$



奇妙的结论

这一错误率还可以通过重复运行降低

理论上，所有的确定性算法必然得到正确的结果，而随机算法却可能出错

但是实践中确定性算法不一定就能够得到正确结果，计算机硬件可能会出错，程序运行时间越长，硬件出错的可能性越大。

结论：一个快速的随机算法会比一个慢速的确定性算法更可靠