第二章 分治法(Divide and Conquer)

1．概述

分治法是广为人知的一种算法设计技术。顾名思义，分治法是分而治之的方法（行政制度就是一种典型的分治法），是指对于一个输入规模为n的问题，采用某种方式把输入分割成k（$1<k\le n$）个子集，从而产生k个不同的可分别求解的同类型问题，解出这k个子问题后，再用适当的方法将其组合成原问题的解（当输入规模n较小时，可直接求解），据此思路设计的算法很自然地可用一个递归过程来表示。

procedure $\text{Solve}(I)$

$n \leftarrow \text{size}(I)$

if$(n \le \text{smallsize})$) then $\text{solution} \leftarrow \text{Directlysolve}(I)$

else

{

divide I into $I_1, \cdots, I_k$

for $i \leftarrow 1$ to k do

$S_i \leftarrow \text{Solve}(I_i)$

repeat

$\text{solution} \leftarrow \text{Combine}(S_1, \cdots, S_k)$

}

endif

return(solution)

end Solve

借助分治法，产生了许多高效算法，众多实际可行的$O(n\log n)$时间复杂度的算法是利用分治法设计的。

我们先从简单例子讲起。

1.1 典型例子选介

例 2.1 整数乘法问题

设X和Y是两个n位的二进制整数，按照乘法规则将X和Y直接相乘，需要$\Theta(n^2)$次"位运算"。使用分治法可将其降低到$\Theta(n^{\log 3})(\log 3 \approx 1.59)$。

为简化讨论，假定n是2的幂，我们分别将X和Y按位均分成两部分，如下图所示：

X: | A | B |　　$X = A2^{n/2} + B$

Y: | C | D |　　$Y = C2^{n/2} + D$

图 2.1

其中，A, B, C, D 都是$n/2$位的二进制整数，X与Y的乘积可表示为

$$XY = (A2^{n/2} + B)(C2^{n/2} + D) = AC2^n + (AD + BC)2^{n/2} + BD \tag{2.1}$$

若依照式(2.1)分而治之，得到四个子问题：AC, AD, BC, BD, 此外，还有一些加法和移位运算。不难看出，加法与移位所耗费的位

运算为$\Theta(n)$，因此，若设两个n位数相乘的时间复杂度（可以认为是数据无关的)为$T(n)$, 则

$$\begin{cases} T(1)=1, \\ T(n)=4T(n/2)+\Theta(n),\ n\geq 2 \end{cases} \tag{2.2}$$

设 $n=2^k$, 则

$$T(n)=4T(2^{k-1})+c2^k=4(4T(2^{k-2})+c2^{k-1})+c2^k$$

$$=4^2T(2^{k-2})+c4\cdot 2^{k-1}+c2^k=\cdots=4^kT(1)+c\sum_{i=1}^{k}4^{k-i}\cdot 2^i$$

$$=4^k+c4^k\sum_{i=1}^{k}(1/2)^i=\Theta(n^2)。 \tag{2.3}$$

时间复杂度并没降低，但将下式

$$AD+BC=(A-B)(D-C)+AC+BD \tag{2.4}$$

代入(2.1)中，得到：

$$XY=AC2^n+\{(A-B)(D-C)+AC+BD\}2^{n/2}+BD \tag{2.5}$$

将4次乘法降低为3次乘法，而加，减和移位次数仍为cn,于是由式(2.5)确定的算法时间复杂度满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=3T(n/2)+\Theta(n),\ n\geq 2 \end{cases} \tag{2.6}$$

其解为（仍设 $n=2^k$）

$$T(n)=3^k+c3^k\sum_{i=1}^{k}(2/3)^i=\Theta(n^{\log 3})\ \ (3^k=2^{k\log 3}) \tag{2.7}$$

从而降低了时间复杂度。

该算法可写成下列递归过程:

算法 2.1 MTBN (Multiplication of Two Bit Number)算法

procedure $\mathrm{MTBN}(X,Y,n)$

//X 和Y 是绝对值小于2^n 的整数，n 是2的幂，函数值等于 X 乘Y //

integer $S, A, B, C, D, M_1, M_2, M_3$

//S 存放 XY 的符号， A, B分别存放 X 的左半部分和右半部分,//

//C, D分别存放Y 的左半部分和右半部分,M_1, M_2, M_3分别//

//存放三个乘积 AC, $(A-B)(D-C)$, BD .//

$S \leftarrow \mathrm{sign}(X)\mathrm{sign}(Y)$

$X \leftarrow |X|$

$Y \leftarrow |Y|$

if $n=1$ then

if($X=1$)and($Y=1$) then return(S)

else return(0)

endif

else

$A \leftarrow X$的左边$n/2$位

$B \leftarrow X$的右边$n/2$位

$C \leftarrow Y$的左边$n/2$位

$D \leftarrow Y$的右边$n/2$位

$M_1 \leftarrow \mathrm{MTBN}(A, C, n/2)$

$M_2 \leftarrow \text{MTBN}(A-B,\ D-C,\ n/2)$

$M_3 \leftarrow \text{MTBN}(B,\ D,\ n/2)$

return($S(M_1 2^n + (M_1 + M_2 + M_3)2^{n/2} + M_3)$)

endif

end MTBN

在此例中，分割成的子问题大小相等，实际上这是设计算法策略中的一条重要原则------平衡.而且，分治法还常需辅之以代数变换才能得到高效算法。下面再举两例，进一步阐明这些思想。

例 2.2 矩阵乘法问题

设A, B是两个n阶方阵，则$C = AB$也是n阶方阵，C中元素

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j) \quad ,(1 \le i,\ j \le n) \qquad (2.8)$$

依此定义式直接计算C的算法时间复杂度为 $\Theta(n^3)$.

分治法提供矩阵相乘的另外一种方法。仍限定n是2的方幂，分别划分A和B为4个$n/2$阶方阵

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad (2.9)$$

则A和B的积为

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \qquad (2.10)$$

如果$n = 2$, 则式(2.10)将对矩阵的元素直接进行计算；当$n > 2$

时，C可通过计算$n/2$阶方阵的乘积与和运算得到。因为n是2的方幂，因而方阵的乘积可递归计算。

就(2.10)式而言, 分成的子问题共有8个$n/2$阶方阵相乘，此外，两个$n/2$阶方阵相加可在cn^2时间内完成，因此这个思路设计的算法时间复杂度$T(n)$满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=8T(n/2)+\Theta(n^2),\ n\geq 2 \end{cases}$$

该递归式的解是：$T(n)=\Theta(n^3)$, 相较于直接计算，没有任何改进。由于方阵的乘法复杂度比加减法高（$\Theta(n^3)$对$\Theta(n^2)$），如果能通过增加加减法的次数以减少乘法的次数，算法的时间复杂度将会得以改善。1969 年，Volker Strassen 发现了一种方法，用7个乘法和18个加减法实现了上述设想，具体步骤如下：

Step1: 记

$$M_1=(A_{11}+A_{22})(B_{11}+B_{22}),$$
$$M_2=(A_{11}-A_{21})(B_{11}+B_{12}),$$
$$M_3=(A_{12}-A_{22})(B_{21}+B_{22}),$$
$$M_4=(A_{11}+A_{12})B_{22},$$
$$M_5=A_{11}(B_{12}-B_{22}),$$
$$M_6=A_{22}(B_{21}-B_{11}),$$
$$M_7=(A_{21}+A_{22})B_{11}.$$

Step2: 通过$M_i(1\leq i\leq 7)$的加减法计算各$C_{ij}(1\leq i,\ j\leq 2)$.

$$C_{11}=M_1+M_3-M_4+M_6,$$

$$C_{12}=M_4+M_5,$$

$$C_{21}=M_6+M_7,$$

$$C_{22}=M_1-M_2+M_5-M_7.$$

上述步骤形成算法的时间复杂度$T(n)$满足下列递归式

$$\begin{cases} T(1)=1, \\ T(n)=7T(n/2)+\Theta(n^2),\ n\geq 2 \end{cases} \tag{2.11}$$

解之得（$n=2^k$）：

$$T(n)=7T(2^{k-1})+cn^2=7(7T(2^{k-2})+cn^2/4)+cn^2$$

$$=7^2T(2^{k-2})+(7/4)cn^2+cn^2=\cdots$$

$$=7^kT(1)+cn^2\sum_{i=0}^{k-1}(7/4)^i=7^k+4cn^2/3((7/4)^k-1)$$

$$=(1+4c/3)7^k-4cn^2/3 \qquad (4^k=n^2)$$

$$=\Theta(n^{\log 7}) \qquad (7^k=2^{k\log 7}=n^{\log 7},\ \log 7\approx 2.81)$$

最后需要指出的是，斯特拉森矩阵乘法只有当n相当大时才优于直接矩阵乘法。

注：斯特拉森算法的另一种形式见“计算机算法基础”（余祥宣，崔国华，邹海明，66 页习题 25）：递归求解C_{ij}共用7次乘法和15次加减法。

例 2.3 快速傅里叶变换(FFT)

离散 Fourier 变换 DFT(Discrete Fourier Transform)广泛应用于数字信号处理等工程领域，它的快速算法 FFT 的设计也是基于分治法。

N 维实向量$(a_0,\ a_1,\ \cdots,\ a_{N-1})$的 Fourier 变换定义如下：

$$\hat{a}_m = \sum_{k=0}^{N-1} a_k e^{2mk\pi i/N}\ ,\quad 0 \le m \le N-1$$

称为该向量的频谱。对$(\hat{a}_0,\ \hat{a}_1,\ \cdots,\ \hat{a}_{N-1})$做 Fourier 逆变换得

$$a_m = \frac{1}{N}\sum_{k=0}^{N-1} \hat{a}_k e^{-2mk\pi i/N}\ ,\quad 0 \le m \le N-1。$$

记

$$\omega_N = e^{2\pi i/N},$$

我们有

$$\hat{a}_m = \sum_{k=0}^{N-1} a_k \omega_N^{mk}\ ,\quad 0 \le m \le N-1。$$

构作对应于N 维实向量$(a_0,\ a_1,\ \cdots,\ a_{N-1})$的多项式

$$a(x) = \sum_{k=0}^{N-1} a_k x^k\ ,$$

则

$$\hat{a}_m = a(\omega_N^m).$$

这样，N 维实向量的 Fourier 变换等价于求以向量分量为系数的 $N-1$次多项式在N 个点

$$\omega_N^m\ ,\quad 0 \le m \le N-1$$

处的值。

在实际应用中，N 通常取为2的方幂。记

$$M = N/2,\quad y = x^2,$$

则$a(x)$可改写为

$$a(x)=\sum_{k=0}^{M-1}a_{2k}x^{2k}+\sum_{k=0}^{M-1}a_{2k+1}x^{2k+1}=\sum_{k=0}^{M-1}a_{2k}y^{k}+x\sum_{k=0}^{M-1}a_{2k+1}y^{k}$$

$$=b(y)+xc(y)$$

其中

$$b(y)=\sum_{k=0}^{M-1}a_{2k}y^{k},\quad c(y)=\sum_{k=0}^{M-1}a_{2k+1}y^{k}$$

可看作分别对应于两个M维向量：

$$(a_0,\ a_2,\ \cdots,\ a_{2(M-1)}),\ (a_1,\ a_3,\ \cdots,\ a_{2(M-1)+1})$$

的多项式。因此有：当$0\le j\le M-1$时，

$$a(\omega_N^j)=b(\omega_N^{2j})+\omega_N^j c(\omega_N^{2j})=b(\omega_M^j)+\omega_N^j\cdot c(\omega_M^j)$$

$$(\omega_N^2=\omega_M)$$

$$a(\omega_N^{M+j})=b(\omega_N^{N+2j})+\omega_N^{M+j}c(\omega_N^{N+2j})=b(\omega_M^j)-\omega_N^j\cdot c(\omega_M^j)$$

$$(\omega_N^N=1,\ \omega_N^M=-1)$$

这样将输入规模为N的问题分割成输入规模为$N/2$的两个子问题，照此设计的算法时间复杂度$T(n)$满足下列递归式：

$$\begin{cases}T(1)=1,\\ T(N)=2T(N/2)+\Theta(N),\ N\ge 2\end{cases}\qquad(2.12)$$

其中cN是分割和组合所需的基本操作数，基本操作为加法，乘法及赋值。设$N=2^r$，解此递归式得：

$$T(N)=2T(N/2)+cN=2^2T(N/2^2)+2cN$$
$$=\cdots=2^rT(1)+rcN=\Theta(N\log N)$$

而由定义式直接计算的时间复杂度显然为$\Theta(N^2)$.

由上述步骤，我们可以给出一个快速傅里叶变换的递归算法。

算法 2.2 快速傅里叶变换

procedure $\mathrm{FFT}(N,\ a(x),\ \omega,\ A)$

//多项式$a(x)=a_{N-1}x^{N-1}+\cdots+a_0$，$N$是$2$的方幂，$\mathrm{a}(\omega^{\mathrm{j}})$//

//（$0\le j\le N-1$）置于数组$A(0:N-1)$中，初始调用FFT//

//时，将ω置为ω_N.//

if $N=1$ then $A(0)\leftarrow a_0$

else

{

$M\leftarrow N/2$

$b(x)\leftarrow a_0+a_2x+\cdots+a_{2(M-1)}x^{M-1}$

$c(x)\leftarrow a_1+a_3x+\cdots+a_{2(M-1)+1}x^{M-1}$

call $\mathrm{FFT}(M,\ b(x),\ \omega^2,\ B)$

call $\mathrm{FFT}(M,\ c(x),\ \omega^2,\ C)$

$t\leftarrow 1$

for $j\leftarrow 0$ to $M-1$ do

$A(j)=B(j)+t\cdot C(j)$

$A(M+j)=B(j)-t\cdot C(j)$

$t=\omega\cdot t$

repeat

endif

end FFT

类似地，我们可以得到快速 Fourier 逆变换算法，时间复杂度也是$\Theta(n\log n)$

应用之一：快速计算卷积

定理 2.1 两个实向量$a(0:n-1)$与$b(0:n-1)$的循环卷积$c=a*b$定义为

$$c(i)=\sum_{j=0}^{n-1}a(j)b((i-j)\bmod(n)),\quad 1\le i\le n-1,\quad (2.13)$$

则有

$$\hat{c}=\hat{a}\hat{b}。\quad (2.14)$$

证明：对任意$1\le m\le n-1$，我们有

$$\begin{aligned}\hat{c}(m)&=\sum_{k=0}^{n-1}c(k)e^{2\pi imk/n}\\&=\sum_{k=0}^{n-1}\left(\sum_{j=0}^{n-1}a(j)b((k-j)\bmod(n))\right)e^{2\pi imk/n}\\&=\sum_{k=0}^{n-1}\sum_{j=0}^{n-1}\left(a(j)e^{2\pi imj/n}\right)\left(b((k-j)\bmod(n))e^{2\pi im(k-j)/n}\right)\\&=\sum_{j=0}^{n-1}\left(a(j)e^{2\pi imj/n}\right)\left(\sum_{k=0}^{n-1}b((k-j)\bmod(n))e^{2\pi im(k-j)/n}\right)\end{aligned}$$

$$= \sum_{j=0}^{n-1} \left(a(j) e^{2\pi imj/n} \right) \sum_{k=0}^{n-1} b((k-j) \bmod (n)) e^{2\pi im((k-j) \bmod (n))/n}$$

$$= \sum_{j=0}^{n-1} \left(a(j) e^{2\pi imj/n} \right) \left(\sum_{l=0}^{n-1} b(l) e^{2\pi iml/n} \right)$$

$$= \left(\sum_{j=0}^{n-1} a(j) e^{2\pi imj/n} \right) \left(\sum_{l=0}^{n-1} b(l) e^{2\pi iml/n} \right)$$

$$= \hat{a}(m)\hat{b}(m)。$$

利用式(2.14)和快速 Fourier 变换及逆变换计算循环卷积的算法时间复杂度为$\Theta(n \log n)$。

1.2 master 方法

采用分治法设计的算法时间复杂度$T(n)$总是满足形如：

$$T(n) = aT(n/b) + f(n) \qquad (2.15)$$

的递归式. 式(2.13)表述了(a, b)型分治算法的时间复杂度，(a, b)型分治算法指递归地将输入规模为n的问题分解为a个输入规模为n/b的子问题，$f(n)$表示分解和合成的时间复杂度。下面定理基本解决了式(2.15)的渐近解。

定理 2.2 (Master Theorem) 设常数$a \geq 1$, $b>1$，$f(n)$是定义在非负整数上的非负函数，$T(n)$是由下列递归式定义的非负整数上的非负函数

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

其中n/b表示$\lfloor n/b \rfloor$或$\lceil n/b \rceil$. 则$T(n)$有下面渐近界：

1. 如果

$$f(n)=O(n^{\log_b a-\varepsilon}),$$

其中，ε是某正常数, 则

$$T(n)=\Theta(n^{\log_b a}); \tag{2.16}$$

2. 如果

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

则

$$T(n)=\Theta(n^{\log_b a}\log n); \tag{2.17}$$

3. 如果

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

ε是某正常数, 且存在常数$c<1$, 使得当n充分大时，

$$af(n/b)\le cf(n),$$

则

$$T(n)=\Theta(f(n)). \tag{2.18}$$

注：n并不总是只取b的方幂，当n是一般整数时，递归式中的$aT(n/b)$应替之以

$$a_1T\left(\lfloor n/b \rfloor\right)+a_2T\left(\lceil n/b \rceil\right),\quad a_1+a_2=a。$$

当然，这样的变化不影响定理的结论。

证明：为了对定理的内容获得一个直观的理解，我们先来看n取b的方幂的情形：

$$T(b^k)=aT(b^{k-1})+f(b^k)=a^2T(b^{k-2})+af(b^{k-1})+f(b^k)$$

$$= \cdots = a^k f(1) + \sum_{i=0}^{k-1} a^i f(b^{k-i}) = \Theta(a^{\log_b n}) + \sum_{i=0}^{\log_b n - 1} a^i f(b^{k-i})$$

$$= \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \qquad (2.19)$$

其中最后的等式利用下面数量关系

$$a^{\log_b n} = n^{\log_b a}。$$

基于(2.19)，我们有

1．若 $f(n) = O(n^{\log_b a - \varepsilon})$ (ε 是正常数), 则

$$\sum_{i=0}^{\log_b n - 1} a^i \cdot f(n/b^i) = \sum_{i=0}^{\log_b n - 1} a^i \cdot O((n/b^i)^{\log_b a - \varepsilon})$$

$$= O(n^{\log_b a - \varepsilon}) \sum_{i=0}^{\log_b n - 1} b^{\varepsilon i} = O(n^{\log_b a - \varepsilon}) \cdot \frac{b^{\varepsilon \log_b n} - 1}{b^{\varepsilon} - 1}$$

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

所以，

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

2．若 $f(n) = \Theta(n^{\log_b a})$, 类似地

$$\sum_{i=0}^{\log_b n - 1} a^i \cdot f(n/b^i) = \sum_{i=0}^{\log_b n - 1} a^i \cdot \Theta((n/b^i)^{\log_b a})$$

$$= \Theta(n^{\log_b a}) \sum_{i=0}^{\log_b n - 1} 1 = \Theta(n^{\log_b a} \log n)$$

所以，

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

3. 如果 $f(n)=\Omega(n^{\log_b a+\varepsilon})$ (ε 是正常数), 且存在常数 $c<1$，使得当 n 充分大时，

$$af(n/b)\le cf(n),$$

则

$$f(n)\le \sum_{i=0}^{\log_b n-1} a^i\cdot f(n/b^i)\le \sum_{i=0}^{\log_b n-1} c^i f(n)$$

$$\le\sum_{i=0}^{\infty}c^i f(n)=\frac{f(n)}{1-c}$$

所以，

$$T(n)=\Theta(f(n))。$$

对于任意正整数 n, 以 n/b 取为 $\lceil n/b\rceil$ 为例来考虑，记

$$\begin{cases} n_0=n, \\ n_i=\lceil n_{i-1}/b\rceil,\ i\ge 1 \end{cases}$$

则

$$n_1\le n/b+1,$$

$$n_2\le n_1/b+1\le n/b^2+1/b+1$$

$$n_3\le n_2/b+1\le n/b^3+1/b^2+1/b+1$$

$$\cdots\cdots$$

$$n_i\le n/b^i+1/b^{i-1}+\cdots+1/b+1$$

$$<n/b^i+\sum_{j=0}^{\infty}1/b^j=n/b^i+b/(b-1)$$

另一方面，

$$n_1\ge n/b$$

$$n_2 \geq n_1/b \geq n/b^2$$
$$\cdots\cdots$$
$$n_i \geq n_{i-1}/b \geq n/b^i$$

取$i = \lfloor \log_b n \rfloor$, 则

$$n_{\lfloor \log_b n \rfloor} < n/b^{\lfloor \log_b n \rfloor} + b/(b-1) < n/b^{\log_b n - 1} + b/(b-1)$$
$$= b + b/(b-1) = O(1)$$

从而类似于(2.19)式，我们有

$$T(n) = aT(n_1) + f(n) = a^2 T(n_2) + af(n_1) + f(n)$$
$$= \cdots = a^{\lfloor \log_b n \rfloor} f(n_{\lfloor \log_b n \rfloor}) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n_i)$$
$$= \Theta(n^{\log_b a}) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f(n_i).$$

再利用

$$n_i = \Theta(n/b^i)$$

则可完成定理的证明。

例 2.4 求满足下列递归式的$T(n)$的渐近表示

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

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

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

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

解: (1) 递归式中

$$a=4,\ b=2,\ \log_b a=2,\ \ f(n)=n=O(n^{\log_b a-1/2}),$$

可归入 **master** 定理中的第一种情形，所以

$$T(n)=\Theta(n^{\log_b a})=\Theta(n^2);$$

（2） $f(n)=\Theta(n^{\log_b a})$，可归入 **master** 定理中的第二种情形，所以

$$T(n)=\Theta(n^{\log_b a}\log n)=\Theta(n^2\log n)$$

（3） $f(n)=\Omega(n^{\log_b a+1/2})$，且

$$af(n/b)=4f(n/2)=n^3/2=f(n)/2,$$

可归入 **master** 定理中的第三种情形，所以

$$T(n)=\Theta(f(n))=\Theta(n^3)$$

（4）递归式中

$$a=1,\ b=2,\ \log_b a=0,\ \ f(n)=c\log n$$

不能归入 **master** 定理中的任何一种情形，所以采用定理中的证明方法将递归式展开到底：(先设$n=2^k$)

$$T(n)=T(2^{k-1})+ck=T(2^{k-2})+c(k-1)+ck=\cdots$$

$$=T(1)+c\sum_{i=1}^{k}i=T(1)+ck(k+1)/2=\Theta(\log^2 n)$$

（$k=\log n$）

下面各节，就利用分治法解决检索，选择，排序等问题分别进行相关讨论。

2. 二分检索

问题：给定一个n元表$A(n)$，且表中元素可比较大小，来判定与表中元素同类型的元素x是否在表中出现。假设表$A(n)$已按递增顺序排列：

$$A(1) < A(2) < \cdots < A(n).$$

注：要检索的表中不可能有两元素相同，且该表总要先排序，以提高检索的效率。

算法 2.3 二分检索 (Binary Search)

procedure BINSRCH(A, low, $high$, x, j)

//给定按递增顺序排列的元素表$A(low, high)$($low \geq 1$,//

//$high \geq low-1$; 当$high = low-1$时, $A(low, high)$是虚拟//

//数组，表示空表)本过程旨在判断x是否在表中出现. 若是，置j,//

//使得$A(j) = x$; 否则置$j = 0$. //

integer low, $high$, j, mid

while $low \leq high$ do

$\text{mid} \leftarrow \lfloor (\text{low}+\text{high})/2 \rfloor$

//$\lfloor x \rfloor$表示不大于x的最大整数//

case

: $x < A(mid)$: $high \leftarrow mid - 1$

: $x > A(mid)$: $low \leftarrow mid + 1$

: else: $\text{j} \leftarrow \text{mid}$; return

endcase

repeat

$j \leftarrow 0$

end BINSRCH

下面先来看算法的正确性。

定理 2.3 过程$\mathrm{BINSRCH}(A,\ low,\ high,\ x,\ j)$能正确地运行。

证明：对表中元素数目（即$high-low+1$）进行归纳，当$high-low+1=0$ (即$high=low-1$)时，表A是空表，此时过程不进入 while 循环，j被置成0, 过程终止, 运行正确；假设当$0\le high-low+1<n$ ($n\ge 1$)时，过程能正确运行，来考查$high-low+1=n$时过程的运行情况。

记 mid=$\lfloor (\mathrm{low+high})/2 \rfloor$，则以输入而论，可分成下列三种情况：

(1) $x<A(mid)$，此时，由表的顺序性，问题归结为在表$A(low, mid-1)$中检索x．在这样的输入下，过程$\mathrm{BINSRCH}(A,\ low,\ high,\ x,\ j)$第一次执行到 while 语句时，因$low=high-n+1\le high$, 满足 while 中的条件语句，所以进入 while 循环体，本次循环的结果是将$high$的值更改为$mid-1$, 然后转回 while 语句，j的值完全由过程此后的运行情况决定，即相当于执行过程$\mathrm{BINSRCH}(A,\ low,\ mid-1,\ x,\ j)$．因为$low\le mid\le high$, 得知$0\le mid-1-low+1<high\text{-}low+1=n$, 由归纳假设，过程运行正确；

(2) $x>A(mid)$, 类似(1)可得正确性；

(3) $x = A(mid)$, 则过程执行一次 while 循环，置 $j = mid$ 后结束，执行正确。

综之，过程 $\mathrm{BINSRCH}(A,\ low,\ high,\ x,\ j)$ 总能正确地运行。

下面来分析算法的时间复杂度，基本操作是比较，每次 while 循环包括两次比较，我们不妨把这两次比较看作一次大比较。下面所计算的比较次数是指大比较次数。

记 $n = high - low + 1$, 即表中元素个数. 当 $n = 0$ 时，仅用一方框表示 x 不在表 A 中；当 $n \geq 1$ 时，可用下图来表示在表 $A(low,\ high)$ 中检索 x 的过程：

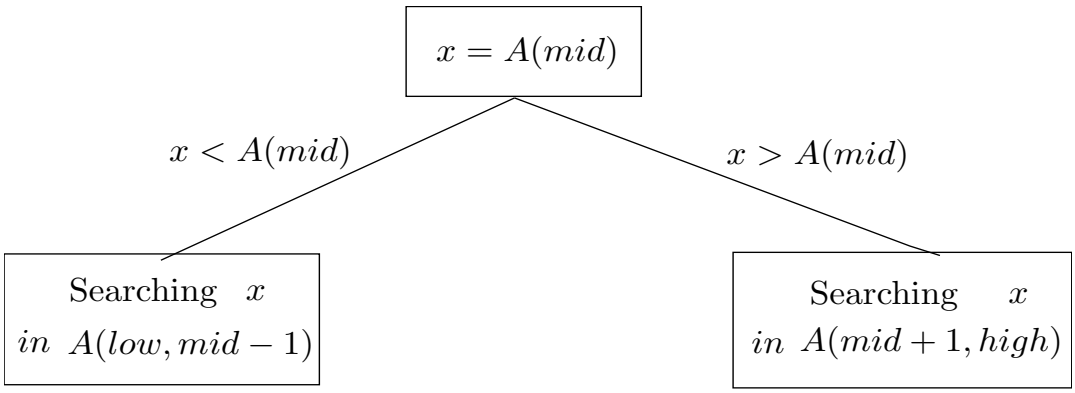

Fig 2.1

由此图结合二元树的递归定义来看，在 $A(low,\ high)$ 中检索 x 的过程 $\mathrm{BINSRCH}(A,\ low,\ high,\ x,\ j)$ 可用一个二元树来表示，此二元树称为二元比较树。

例 2.5 在表 $A(1:7)$, $A(1:9)$ 中检索 x 的算法执行过程对应的二元检索树如下图:

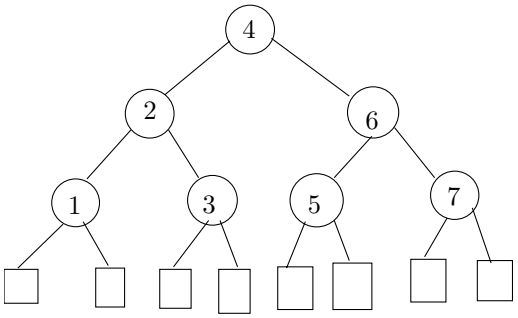

Fig 2.2 (a)

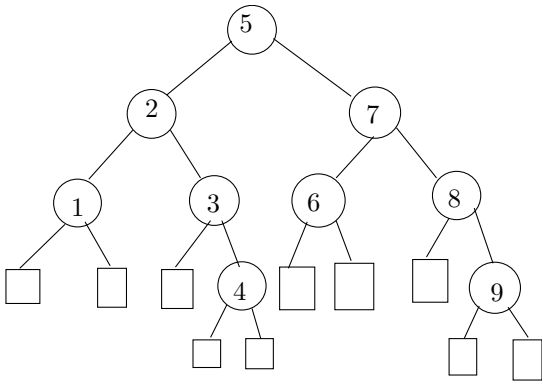

Fig 2.2 (b)

注：1. 二元比较树包含n个内结点（用圆框表示）和$n+1$个外结点（叶子）（用方框表示），内结点的最高级为$\lceil \log(n+1) \rceil$（当$2^k - 1 < n \le 2^{k+1} - 1$时，$\lceil \log(n+1) \rceil = k+1$）,外结点的最高级为$\lceil \log(n+1) \rceil + 1$，外结点只可能在$\lceil \log(n+1) \rceil + 1$级或$\lceil \log(n+1) \rceil$级上。

2. 当$n = 0$时，二元比较树退化为一片叶子，表示表A是空的，x不在空表A中；当$n \ge 1$时，二元比较树的n个内结点表示x可能所在的表A中的n个位置，$n+1$个外结点表示x不在A中的$n+1$种情形。

3. $x = A(i)(1 \le i \le n)$时过程所花费的比较次数，恰是其对应内结点的级数；当$A(i) < x < A(i+1)$ $(0 \le i \le n$, 其中$A(0)$表示虚拟最小值, $A(n+1)$表示虚拟最大值) 时，过程所化费的比较次数，则是表示此种情形的叶子的级数减去1，其值为$\lceil \log(n+1) \rceil$或$\lceil \log\ (n+1) \rceil - 1$.

由以上事实知，算法的最坏时间复杂度为

$$WT(n)=\lceil \log(n+1) \rceil \qquad (2.20)$$

下面来看算法的平均时间复杂度$MT(n)$，假定x在表A中的概率是q，且假设x在A中每个位置出现的概率相同，即x在表A中的n种情况出现的概率都是$1/n$. 记x在表A中的n种情况的平均时间复杂度为$S(n)$；同样记x不在表A中的$n+1$种情况的平均时间复杂度为$U(n)$. 则

$$MT(n)=q\cdot S(n)+(1-q)\cdot U(n) \qquad (2.21)$$

此处，利用了下面概率论中的基本结论之一：

定理 设X，Y是随机变量，则

$$E(X)=\int_y p(Y=y)E(X\mid Y=y)dy.$$

或等价地

$$E(X)=E(g(Y)),$$

其中

$$g(y)=E(X\mid Y=y).$$

显然, 不论x不在表A中的$n+1$种情况的概率分布是什么，都有

$$\lceil \log(n+1) \rceil -1<U(n)\le \lceil \log(n+1) \rceil . \qquad (2.22)$$

下面利用$U(n)$来估算$S(n)$. 需要这样一个事实：含n个内结点的二元树中，内部路径长度I和外部路径长度E满足下列关系：

$$E=I+2n \qquad (2.23)$$

因为结点到根结点的内部路径是结点的级数减 1，所以

$$S(n)=\frac{I}{n}+1\ =\frac{E}{n}-1=(\frac{n+1}{n})(\frac{E}{n+1})-1, \qquad (2.24)$$

由(2.22)知，

$$\lceil \log(n+1) \rceil - 1 < \frac{E}{n+1} \le \lceil \log(n+1) \rceil. \qquad (2.25)$$

将(2.25)代入(2.24) 得：

$$\frac{n+1}{n}\lceil \log(n+1) \rceil - 2 - \frac{1}{n} < S(n) \le \frac{n+1}{n}\lceil \log(n+1) \rceil - 1,$$

化简为（考虑$n \ge 1$）

$$\lceil \log(n+1) \rceil - 2 < S(n) \le \lceil \log(n+1) \rceil. \qquad (2.26)$$

将(2.26)和(2.22)代入(2.21)得到

$$\lceil \log(n+1) \rceil - 2 < MT(n) \le \lceil \log(n+1) \rceil.$$

这说明算法的平均时间复杂度也是$\Theta\left(\lceil \log\ (n+1) \rceil\right)$，且最坏时间复杂度比平均时间复杂度不超过两个基本操作.

下面说明算法 2.3 是以比较为基础的检索算法类中最坏时间复杂度为最佳的算法。只允许进行元素间的比较而不允许对它们实施其它运算的检索算法称为以比较为基础的检索算法。

在一个有序表A中检索x, 比较操作只需在x和表A中的元素之间进行，这种比较可分成两种情况：

其一：

$x = A(j_0)$;

$x < A(j_0)$, 归结为在表$A(low,\ j_0 - 1)$中检索x;

$x > A(j_0)$, 归结为在表$A(j_0 + 1,\ high)$中检索x;

其二：

$x < A(j_0)$, 归结为在表$A(low,\ j_0 \text{-} 1)$中检索x;

$x \geq A(j_0)$, 归结为在表 $A(j_0, high)$ 中检索 x;

或

$x \leq A(j_0)$, 归结为在表 $A(low, j_0)$ 中检索 x;

$x > A(j_0)$, 归结为在表 $A(j_0+1, high)$ 中检索 x;

我们仅对采取第一类比较的算法来进行分析, 采取第二类比较的算法也有类似的结果.

依照对算法 2.3 的分析，任何以比较为基础的算法都可用二元比较树来表示算法执行的过程．这样的二元比较树都含 n 个内结点(每个内结点都表示 x 在表 A 中某个位置时的算法执行终止处), $n+1$ 个外结点(每个外结点都表示 x 在表 A 外某个位置时算法执行终止处), 算法 A 在输入规模为 n 的情况下所化费的最大比较次数 $WT_A(n)$ 等于对应的二元比较树内结点的最高级数.

我们知道，内结点最高级为 k 的二元树含内结点的数目至多为 2^k-1 ，所以 $n \leq 2^k-1$ ，即 $k \geq \lceil \log\ (n+1) \rceil$ ，即 $WT_n \geq \lceil \log\ (n+1) \rceil$, 这说明算法 2.3 在最坏情况下是最佳的.

综之得到：

定理 2.4 设表 $A(n)$ 含有 n 个不同的元素，它们被排序成

$$A(1) < A(2) < \cdots < A(n),$$

又设以比较为基础判断某元素 x 是否在表 $A(n)$ 的任何算法在最坏情况下所需的最小比较次数是 $\mathrm{FIND}(n)$, 那末

$$\mathrm{FIND}(n) \geq \lceil \log\ (n+1) \rceil。$$

3. 找最大和最小元素

问题 1: 在n个不同的元素中找最大(小)元。

以找最大元为例（找最小元是类似的），简单的方法是将元素逐个进行比较，具体步骤可描述如下。

算法 2.4 直接找最大元

Procedure $\mathrm{MAX}(A, n, \max)$

//将$A(1:n)$中的最大元置于$\max$，$n \geq 2$//

integer i, n

$\max \leftarrow A(1)$

for $i \leftarrow 2$ to n do

if $A(i) > \max$ then $\max \leftarrow A(i)$ endif

repeat

end MAX

算法中的基本操作是比较，显然该算法是数据无关的，$T(n) = n - 1$.事实上，该算法在以比较为基础的找最大元算法类中是最优的。

定理 2.5 任何以比较为基础的在n个元素中寻找最大元的算法，必须至少作$n-1$次比较。

证明：算法中的每一次比较只能确定出某一元素小于等于其他元素中的某一个，如果算法包含k次比较，则至多能确定k个元素不是最大元，因此若$k \leq n-2$,则还有$m(m \geq n-k \geq 2)$个元素未发现小于任何其他元素，最大元应在这m个元素中间产生，但算法并不能最终确定这m个元中究竟哪个最大，这说明$k \geq n-1$.

注：也可考虑用连通图中结点数n与边数m的关系: $m \geq n-1$来证明.

但同时找最大元与最小元却并不需要$2(n-1)$次比较。

算法 2.5 直接找最大和最小元素

```
procedure MAXMIN(A, n, max, min)
//将A(1:n)中的最大元素置于 max, 最小元置于 min, n≥2//
integer i, n
max ← min ← A(1)
for i ← 2 to n do
   if A(i) > max then max ← A(i)
   else
        if A(i) < min then min ← A(i) endif
   endif
repeat
end MAXMIN
```

注：在循环体的第一个 if 语句中，只有当$A(i) < \max$时，才可能有$A(i) < \min$, 所以关于 min 寻找之比较可嵌入其中。

该算法的最好情况在元素按递增顺序排列时出现，元素比较数是$n-1$, 最坏情况在元素按递减次序排列时出现，元素比较数是$2(n-1)$. 至于在平均情况下，即假设$n!$种排列具有相同概率的情况下，平均比较次数是

$$\sum_{i=2}^{n}(1/i+2(1-1/i))=2(n-1)-H(n)+1=2(n-1)-\ln\ n+\alpha_n$$

其中$H(n)$为n阶调和数：

$$H(n)=\sum_{i=1}^{n}1/i=\ln\ n+E_n,$$

$$\alpha_n=1-E_n,$$

E_n有极限，令

$$E=\lim_{n\to\infty}E_n$$

E是欧拉常数,

$$E\approx 0.577.$$

注：比较数设为随机数$x(n)$，则平均比较次数为$E(x(n))$．设第i次循环需作的比较数为随机数$x^i(n)$，则

$$x(n)=\sum_{i=2}^{n}x^i(n), \qquad (2.27)$$

从而，

$$E(x(n))=\sum_{i=2}^{n}E(x^i(n)) \qquad (2.28)$$

又因为

$$E(x^i(n))=1/i+2(1-1/i) \qquad (2.29)$$

将(2.29)代入(2.28)得

$$E(x(n))=\sum_{i=2}^{n}(1/i+2(1-1/i)).$$

下面用分治法来设计一个算法，基本考虑是将任一较大实例：

$$I=(n,\ A(1),\ \cdots,\ A(n)) \quad (\mathrm{n}\geq 3)$$

分成两个实例：

$$I_1=\left(\lceil n/2\rceil;\ A(1),\ \cdots,\ A\left(\lceil n/2\rceil\right)\right)$$

和

$$I_2=\left(\lfloor n/2\rfloor;\ A\left(\lceil n/2\rceil+1\right),\ \cdots,\ A(n)\right)$$

则

$$\max(I)=\max\{\max(I_1),\ \max(I_2)\},$$

$$\min(I)=\min\{\min(I_1),\ \min(I_2)\}$$

而当$n\leq 2$时，不作任何分割直接得到其解。

算法 2.6 递归求取最大和最小元素

procedure $\mathrm{MAXMIN}(i,\ j,\ \max_1,\ \min_1)$

// $A(1:n)$是含有n个元素的数组，参数i, j是整数, $1\leq i\leq j\leq n$,//

//该过程把$A(1:n)$中的最大和最小元素分别赋给$\max_1$和$\min_1$. //

integer i, j; global n, $A(1:n)$

case

$:i=j:\ \max_1\leftarrow\min_1\leftarrow A(i)$

$:i=j-1:$ if $A(i)<A(j)$

then $\max_1\leftarrow A(j),\ \min_1\leftarrow A(i)$

else $\max_1\leftarrow A(i),\ \min_1\leftarrow A(j)$

endif

: else: $mid\leftarrow\lfloor(i+j)/2\rfloor$

call $\mathrm{MAXMIN}(i,\ mid,\ \max_2,\ \min_2)$

call $\mathrm{MAXMIN}(mid+1,\ j,\ \max_3,\ \min_3)$

$\max_1 \leftarrow \max(\max_2,\ \max_3)$

$\min_1 \leftarrow \min(\min_2,\ \min_3)$

endcase

end MAXMIN

这个过程最初由 call$\mathrm{MAXMIN}(1,\ \mathrm{n},\ \max,\ \min)$所调用。

分析比较数: 该过程所需比较数$T(n)$ ($n = j - i + 1$)满足下列递归式:

$$\begin{cases} T(1)=0,\ T(2)=1, \\ T(n)=T\left(\lfloor n/2 \rfloor\right)+T\left(\lceil n/2 \rceil\right)+2,\ n\geq 3 \end{cases} \tag{2.30}$$

当n是 2 的幂时，

$$T(n)=\lceil 3n/2 \rceil - 2. \tag{2.31}$$

可证明如下：记$n=2^k$. 当$k=1$时，$T(2)=1$, 等式成立；假设$k=k_0$ ($k_0\geq 1$)时, 等式成立，即$T(2^{k_0})=3\cdot 2^{k_0-1}-2$, 来看$k=k_0$时的情况:

$$T(2^{k_0+1})=2T(2^{k_0})+2=2(3\cdot 2^{k_0-1}-2)+2=3\cdot 2^{k_0}-2$$

等式仍成立。证毕。

但当n不是 2 的幂时，(2.31)不一定成立. 例如: $T(6)=8$, $\lceil 3\times 6/2 \rceil - 2 = 7$.

(2.31)中的"="替换成"≥", 就对任何整数成立了。稍后我们将会证明，以元素比较为基础的找最大和最小的算法，在最坏情况下元

素比较下界为$\lceil 3n/2 \rceil - 2$.

所以该算法就元素比较数而言是比较好的（甚至当n是 2 的幂时，在最坏情况下是最佳的），但由于递归算法中，i，j，max，min 进出栈（栈深度为$\lceil \log n \rceil$）所带来的开销，再考虑到i，j间比较与元素比较时间相类时，此算法可能反而比直接比较算法花销的时间更多.

下面另外给出一个用分治策略设计的算法，基本思想是将实例：

$$I = (n;\ A(1),\ \cdots,\ A(n))\ \ (n > 2)$$

分割成$\lfloor n/2 \rfloor$个实例:

$$I_i = (2;\ A(i),\ A(n+1-i))\ \ (1 \le i \le \lfloor n/2 \rfloor),$$

分别将其中的最大元和最小元赋给$A(i)$和$A(n+1-i)$. 然后直接求

$$\mathrm{A}(1),\ \cdots,\ \mathrm{A}\left(\lceil \mathrm{n}/2 \rceil\right)$$

中的最大元和

$$A\left(n+1-\lceil n/2 \rceil\right)\ \cdots,\ A(n)$$

中的最小元，就分别得到实例I的最大元和最小元。

算法 2.7 非递归分治求取最大和最小元素

```
procedure BMAXMIN(A, n, max, min)
integer i, n, N
if n = 2 then
        if A(1) > A(2) then max ← A(1); min ← A(2)
                       else max ← A(2); min ← A(1)
        endif
```

else

$N \leftarrow \lfloor n/2 \rfloor$

for $i \leftarrow 1$ to N do

if $A(i) < A(n+1-i)$

then interchange$(A(i),\ A(n+1-i))$

endif

repeat

$N \leftarrow \lceil n/2 \rceil;\ \max \leftarrow A(1);\ \min \leftarrow A(n)$

for $i \leftarrow 2$ to N do

if $A(i) > \max$ then $\max \leftarrow A(i)$ endif

if $A(n+1-i) < \min$ then $\min \leftarrow A(n+1-i)$ endif

repeat

endif

end BMAXMIN

下面来说明该算法元素比较数$T(n) = \lceil 3n/2 \rceil - 2$.

证明：当$n = 2$时，等式成立；当$n > 2$时

$$\begin{aligned} T(n) &= \lfloor n/2 \rfloor + 2\left(\lceil n/2 \rceil - 1\right) \\ &= \left(\lfloor n/2 \rfloor + \lceil n/2 \rceil\right) + \lceil n/2 \rceil - 2 \\ &= n + \lceil n/2 \rceil - 2 \\ &= \lceil 3n/2 \rceil - 2 \end{aligned}$$

该算法的比较数达到了最坏情况下的下界，且不需要递归，即没有进出栈所带来的开销，因此应比前面两算法效率高，可说是最佳算

法。

定理 2.6 任何以比较为基础求n个元素中最大和最小元的算法在最坏情况下至少需要$\lceil 3n/2 \rceil - 2$次比较。

证明：我们以一个四元组(a, b, c, d)来表示算法执行过程中的状态，其中，a是尚未参加过比较的元素数，b是在以往的比较中从未输过的元素数，c是在以往的比较中从未赢过的元素数，d是既输过也赢过的元素数。算法以状态$(n, 0, 0, 0)$起始，而应以$(0, 1, 1, n-2)$结束。状态(a, b, c, d)下进行一次比较而使状态发生变化成为下面五种情形（比较坏的）之一：

(1) $(a-2, b+1, c+1, d)$: 如果$a \geq 2$且a中两元素进行比较；

(2) $(a-1, b, c+1, d)$: $a \geq 1$, a中一元素与b中一元素进行比较, 结果b中元素赢;

(3) $(a-1, b+1, c, d)$: $a \geq 1$, a中一元素与c中一元素进行比较, 结果a中元素赢;

(4) $(a, b-1, c, d+1)$: $b \geq 2$, b中两元素进行比较;

(5) $(a, b, c-1, d+1)$: $c \geq 2$, c中两元素进行比较。

{ (6) (a, b, c, d): $b \geq 1$, $c \geq 1$, b中一元素与c中一元素比较，结果b中元素赢。状态不发生变化}

为了达到状态$(0, 1, 1, n-2)$, a必须由n变为0, 只有(1), (2)或(3)三种比较对此作贡献，这至少需要$\lceil n/2 \rceil$次比较；而d也必须由0变为$n-2$, 只有(4)或(5)两种比较对此作贡献，这至少需要$n-2$次比

较，所以总共至少需要

$$\lceil n/2 \rceil + n - 2 = \lceil 3n/2 \rceil - 2$$

次比较。

4. 排序

所谓排序(sorting)就是将一些可比较大小的对象按递增或递减的顺序进行排列。这类问题在计算机的处理工作中经常遇到。据早先统计，包括所有计算机的用户在内，在它们的计算机上，运行时间的四分之一以上是花在排序上，所以排序问题引起众多计算机专家的关注，陆续产生了一系列排序算法。

4.1 排序算法概述

第一个排序的计算机程序是由 Von Neumann 写于 1946 年，并在第一台计算机上运行，但真正开始研究排序算法是本世界 50 年代的事。最简单直观的排序算法是选择排序，冒泡排序和插入排序，但是，它们的运行时间都是$\Theta(n^2)$. 为提高排序速度，人们不断改进上述算法。1954 年，D. L. Shell 提出了缩小增量法，这是对插入排序的有效改进。1962 年，C.A.R hoare 提出了快速排序(quicksort)方法，它的平均时间复杂度是$\Theta(n\log n)$，而最坏情况下运行时间仍是$\Theta(n^2)$.1964 年，由 J. Williams 首先提出而后经 Floyd 改进的堆排序算法是一个既在平均时间又在最坏情况下都是$\Theta(n\log n)$的有效算法。这期间人们还提出了一些别的排序算法，如归并排序，基数排序等成熟的算法。并且，在这期间，人们也认识到了比较排序的时

间下界是$\Theta(n\log n)$.

到了 70 年代，排序方法的改进就更精细了，Singeton 在 1969 年，Frazer 和 Mckellar 在 1970 年提出了对快速排序的改进. Blum, Floyd, Pratt, Rivest 和 Tarjan 给出了$O(n)$时间的求顺序统计量（一般选择问题）的算法。随后 Floyd 和 Rivest 又在 1973 年对此作了改进。

稳定的排序算法是 Horvath 在 1974 年提出的，大多数简单算法是稳定的，而多数知名算法是不稳定的。迄今所知的$\Theta(n\log n)$的原地且又稳定性的算法见朱洪有关文章。

1973 年, D. E. Knuth 在他的名著"the art of computer programming"第三卷（有中译本）中，对排序算法做了全面总结，使排序算法更系统，更完善。

最后，还要提及的是，1980 年, D. E. Knuth 的学生 Sedgewick 在他的博士论文"Quicksort"中，对排序算法作了全面，精细的分析，他的分析是对一个计算机算法进行数学分析的典范。

排序（sorting)是计算机程序中的一种重要运算，它的功能是将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。

注：为查找方便，通常希望计算机中的表是按关键字有序的。

为便于讨论，在此首先要对排序下一个确切的定义。

定义 2.1 假设含n个记录的序列为

$$\{R_1, R_2, \cdots, R_n\}$$

其相应的关键字序列为

$$\{k_1, k_2, \cdots, k_n\}$$

需要确定$1, 2, \cdots, n$的一种排列$p_1, p_2, \cdots, p_n$，使关键字序列满足如下的非递减关系：

$$k_{p_1} \le k_{p_2} \le \cdots \le k_{p_n}$$

从而把记录序列重排为按关键字有序的序列

$$\{R_{p_1}, R_{p_2}, \cdots, R_{p_n}\}$$

这样的一种操作称为排序。

为称谓方便，在不引起混淆的情况下，我们将把关键字的有序性等同于记录或数据元素本身的有序性。

若n元序列$\{a_1, a_2, \cdots, a_n\}$中有相同的元素，则排序结果是不唯一的。当然，任何排序算法只给出一种结果，我们希望排序算法具有下面定义的稳定性。

定义 2.2 对任意$n>1$和任意n元序列$\{a_1, a_2, \cdots, a_n\}$,排序算法$A$总能给出关于$1, 2, \cdots, n$的排列$P$，使其满足

(1) $a_{p_i} \le a_{p_{i+1}}$（$1 \le i \le n-1$）；

(2) 当$a_{p_i} = a_{p_j}, i < j$时，必有$p_i < p_j$，即序列中任意两个相同元素排序前后的顺序关系保持不变.

则称排序算法是稳定的。

注：稳定排序算法的提法背景，含多个关键字（具有优先顺序）的记录排序问题。

由于待排序的记录规模不同，使得排序过程中涉及的存储器不

同，可将排序方法分为两大类：一类是内部排序，指的是待排序记录存放在计算机随机存储器中进行的排序过程；另一种是外部排序，指的是待排序记录的数量太大，以致内存不能一次容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

本节集中讨论内部排序。

内部排序方法基本可分为以下几类：

一. 插入排序

1. 直接插入

2. 折半插入（二分插入）

3. 两路插入

4. 歇尔方法

二. 交换排序

1. 冒泡排序

2. 快速排序

三. 选择排序

1. 直接选择

2. 树选择

3. 堆选择

四. 归并排序

五. 分布排序

1. 基数排序

2. 映射排序

4.1 插入排序

最直接的排序方法是插入法，算法具体描述如下：

算法 2.8 插入排序

procedure INSERTISORT(A, n)

// 将$A(1:n)$中的元素按非降次序排列, $n \geq 1$.//

integer i, j

for $j \leftarrow 2$ to n do //$A(1: j-1)$已排好序//

$item \leftarrow A(j)$; $i \leftarrow j-1$

While ($i > 0$ and $item < A(i)$) do //$0 \leq i < j$//

$A(i+1) \leftarrow A(i)$; $i \leftarrow i-1$

repeat

$A(i+1) \leftarrow item$

repeat

end INSERTISORT

算法的基本操作：赋值与比较。while 体中的语句可能执行0次到$j-1$次，而j从2变到n, 因此，这过程的最坏情况限界是

$$3(n-1)+\sum_{j=2}^{n}2(j-1)=(n+3)(n-1);$$

如果输入数据本来就是非降次序排列的，则根本不会进入 while 的循环体，这是最好的情况，基本操作次数是：$3n-3$.

下面来分析其平均时间复杂度：仅考虑要排序的集合$S=\{a_1, a_2, \cdots, a_n\}$中的元素均不相同的情况，由于将$S$输入时

其元素是按某种顺序排列在数组$A(1:n)$中的，这种排列方式共有$n!$种，假定所有排列方式等概出现。对排列

$$A(1),\ A(2),\ \cdots,\ A(n)$$

定义如下反序概念：若$i<j$，但$A(i)>A(j)$，则称$\{(i,\ A(i)),\ (j,\ A(j))\}$是$A$的一个反序，设$A$的反序数是$I(A)$，则插入算法在输入$A$的情况下的基本操作数是：$3n-2+2I(A)$).

注意到A的反排列A^{-1}:

$$A(n),\ A(n-1),\ \cdots,\ A(1)$$

的反序数$I(A^{-1})$与$I(A)$的和为$n(n-1)/2$. 这是因为S中任二元素$A(i),\ A(j)$要么在排列A中构成反序

$$\{(i,\ A(i)),\ (j,\ A(j))\},$$

要么在排列A^{-1}中构成反序

$$\{(n+1-i,\ A(i)),\ (n+1-j,\ A(j))\},$$

且二者仅有其中之一成立。于是插入算法的平均时间复杂度

$$MT(n)=\frac{2}{n!}\sum_{A\in S_n} I(A)+3n-3, \qquad (2.32)$$

其中S_n是S中n个元素所有不同排列构成的集合。同时

$$MT(n)=\frac{2}{n!}\sum_{A\in S_n} I(A^{-1})+3n-3, \qquad (2.33)$$

(2.32)+(2.33)得到

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n}(I(A)+I(A^{-1}))+3n-3$$
$$=\frac{1}{n!}\sum_{A\in S_n}n(n-1)/2+3n-3$$
$$=n(n-1)/2+3n-3$$
$$=(n-1)(n+6)/2$$

尽管直接插入法在最坏情况和平均情况下的时间复杂度都是$\Theta(n^2)$，由于其简单性，当n相当小或输入的元素列几乎有序的情况下，采用这种算法还是相当理想的。

二分检索树插入法：第一个元素$A(1)$作为单结点的树T_1，以下列递归方法生成$A(1:j)$($j\geq 2$)中元素的排序树T_j:

(1) 若$A(j)<T_{j-1}$的根结点，则(a) 若T_{j-1}无左子树，则将$A(j)$作为其左子树； (b) 否则，以递归方法将$A(j)$插入其左子树;

(2) 若$A(j)\geq T_{j-1}$的根结点, 则(a) 若T_{j-1}无右子树，则将$A(j)$作为其右子树； (b) 否则，以递归方法将$A(j)$插入其右子树。

注：该排序算法是稳定的。

例：$(1,\ 2,\ 3)$，$(2,\ 1,\ 3)$，$(1,\ 3,\ 2)$，$(3,\ 1,\ 2)$,$(2,\ 3,\ 1)$，$(3,\ 2,\ 1)$分别对应的二分检索树如下图:

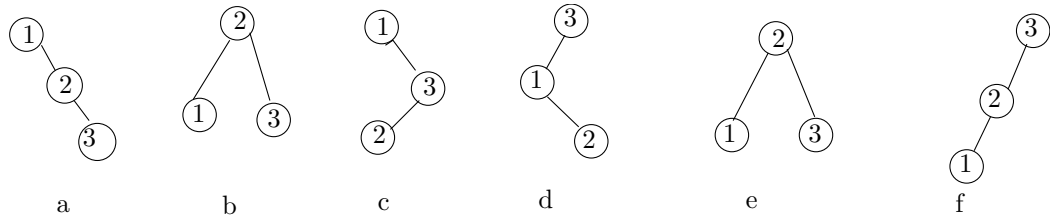

Fig 2.3

每次插入一个结点x的花费(比较次数)是该结点到根的路径长度。

算法的最坏时间复杂度:

$$WT(n)=0+1+2+\cdots+n-1=n(n-1)/2.$$

如果S中元素各不相同，在$n!$种输入中，总共有2^{n-1}种输入达到最坏时间复杂度。

下面来计算算法的平均时间复杂度$MT(n)$。同样假设S中元素各不相同，不妨假定：$a_1<a_2<\cdots<a_n$，并设这n个元素的每个排列$A(1:n)$（总共$n!$个）作为输入数组出现的概率相同。记这些排列组成的集合为S_n。则算法的平均时间复杂度可表示为

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n} i(B(A)), \tag{2.34}$$

其中$i(B(A))$表示由$A(1:n)$中元素生成的排序树的各结点到根的路径长度总和。

下面来设法得到关于$MT(n)$的递归关系式。为表示方便，约定

$$MT(0)=0, \tag{2.35}$$

当$n>0$时，

$$i(B(A))=n-1+i(L(A))+i(R(A)),$$

因此，

$$MT(n)=\frac{1}{n!}\sum_{A\in S_n} i(B(A))$$

$$=\frac{1}{n!}\sum_{A\in S_n}(n-1+i(L(A))+i(R(A)))$$

$$=n-1+\frac{1}{n!}\sum_{A\in S_n} i(L(A))+\frac{1}{n!}\sum_{A\in S_n} i(R(A))$$

$$=n-1+\frac{2}{n!}\sum_{A\in S_n} i(L(A)) \qquad (2.36)$$

设S_n^j是使$B(A)$的根为S中第j小元素的那些排列$A(1:n)$组成的集合，即

$$S_n^j=\{A\in S_n : A(1)=a_j\}$$

则

$$S_n=\bigcup_{j=1}^{n} S_n^j \qquad (2.37)$$

并且, 当$i\neq j$时

$$S_n^i\cap S_n^j=\varnothing \qquad (2.38)$$

当$A\in S_n^j$时，$L(A)$完全由前$j-1$个元素$a_1,\cdots,a_{j-1}$在A中相对排列方式决定，每个固定的这种相对排列方式对应着S_n^j中的

$$C_{n-1}^{j-1}\cdot(n-j)!=(n-1)!/(j-1)!$$

种排列方式。设S_{j-1}是$a_1,\cdots,a_{j-1}$的$(j-1)!$种排列组成的集合，对任意$\sigma\in S_{j-1}$，记

$$S_n^{j,\sigma}:=\{A\in S_n^j:\ a_1,\cdots,a_{j-1}\text{在}A\text{中的相对排列}=\sigma\}$$

则

(1) $S_n^j = \bigcup_{\sigma \in S_{j-1}} S_n^{j,\sigma}$,

(2) 当$\sigma_1, \sigma_2 \in S_{j-1}$，$\sigma_1 \neq \sigma_2$时，

$$S_n^{j,\sigma_1} \bigcap S_n^{j,\sigma_2} = \varnothing.$$

(3) 对于任意$\sigma \in S_{j-1}$，都有

$$| S_n^{j,\sigma} | = (n-1)!/(j-1)!.$$

因此

$$\sum_{A \in S_n^{j,\sigma}} i(L(A)) = \sum_{A \in S_n^{j,\sigma}} i(B(\sigma)) = (n-1)!/(j-1)! i(B(\sigma))$$

从而

$$\sum_{A \in S_n^j} i(L(A)) = \sum_{\sigma \in S_{j-1}} \sum_{A \in S_n^{j,\sigma}} i(L(A))$$

$$= \sum_{\sigma \in S_{j-1}} (n-1)!/(j-1)! i(B(\sigma))$$

$$= (n-1)!/(j-1)! \sum_{\sigma \in S_{j-1}} i(B(\sigma))$$

$$= (n-1)! MT(j-1) \qquad (2.39)$$

由(2.36—2.39)得到: 当$n > 0$时，

$$MT(n) = n - 1 + \frac{2}{n!} \sum_{j=1}^{n} \sum_{A \in S_n^j} i(L(A))$$

$$= n - 1 + \frac{2}{n!} \sum_{j=1}^{n} (n-1)! MT(j-1)$$

$$= n - 1 + \frac{2}{n} \sum_{j=1}^{n} MT(j-1) \qquad (2.40)$$

下面来解此递归式。将式(2.38)变形为

$$n \cdot MT(n) = n(n-1) + 2\sum_{j=1}^{n} MT(j-1)$$

分别以$n = k$，$n = k-1$代入上式得

$$k \cdot MT(k) = k(k-1) + 2\sum_{j=1}^{k} MT(j-1) \qquad (2.41)$$

$$(k-1) \cdot MT(k-1) = (k-1)(k-2) + 2\sum_{j=1}^{k-1} MT(j-1) \qquad (2.42)$$

(2.41)–(2.42)得

$$k \cdot MT(k) - (k-1)MT(k-1) = 2(k-1) + 2MT(k-1)$$

整理且令$k = n$得

$$n \cdot MT(n) = (n+1) \cdot MT(n-1) + 2(n-1) \qquad (2.43)$$

上式两边同除以$2n(n+1)$得

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

$$= \frac{MT(n-1)}{2n} + \frac{2}{n+1} - \frac{1}{n} \qquad （2.44）$$

反复运用（2.44）得到

$$\frac{MT(n)}{2(n+1)} = \frac{MT(0)}{2} + \left(\frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{2}\right) - \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right)$$

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

$$= \ln\ n + E_n - \frac{2n}{n+1}$$

最后得到

$$MT(n) = 2(n+1)\ln\ n - 4n + (n+1)E_n \qquad (2.45)$$

用分治策略来设计排序算法，可得到两种排序算法：归并排序(Mergesort)，快速排序(Quicksort)。

这两种排序的主要特征分别是

归并排序(Mergesort): 分割易(easy division), 组合难(hard combination);

快速排序(Quicksort): 分割难(hard division), 组合易(easy combination).

4.2 归并排序

归并排序可使最坏时间复杂度变为$\Theta(n\log n)$。其基本思想是：将 $A(1), \cdots, A(n)$ 分成两个集合 $A(1), \cdots, A(\lfloor (n+1)/2 \rfloor)$ 和 $A(\lfloor (n+1)/2 \rfloor + 1), \cdots, A(n)$, 对每个集合分别归并排序，然后将已排好序的两个序列归并成一个有序的序列。

算法 2.9 归并排序

procedure MERGESORT($A, low, high$)

// $A(low: high)$是一个全程数组，它含有$high - low + 1 \geq 1$ 个//

//待排序的元素//

integer $low, high, mid$

if ($low < high$) then $mid \leftarrow \lfloor (low + high)/2 \rfloor$ //分割点 //

call MERGESORT(A, low, mid)

//子集合排序//

call MERGESORT($A,\ mid+1,\ high$)

//另一子集合排序//

call MERGE($A,\ low,\ mid,\ high$) //归并//

endif

end MERGESORT

算法 2.10 使用辅助数组归并两个已排序的集合

Procedure MERGE($A,\ low,\ mid,\ high$)

//$A(low,\ high)$是一个全程数组，它含有两个分别放在//

//$A(low,\ mid)$和$A(mid+1,\ high)$中已排好序的子集合. 目标//

//是将这两个已排好序的集合按顺序归并成序列并存放到//

//$A(low,\ high)$.使用辅助数组$B(low,\ high)$.//

integer $h,\ i,\ j,\ k,\ low,\ mid,\ high$

global $A(low:\ high)$; local $B(low,\ high)$

$h \leftarrow low;\ i \leftarrow low;\ j \leftarrow mid+1$

while $i \le mid$ and $j \le high$ do //当两个集合都没取尽//

if $A(i) \le A(j)$ then $B(h) \leftarrow A(i);\ i \leftarrow i+1$

else $B(h) \leftarrow A(j);\ j \leftarrow j+1$

endif

$h \leftarrow h+1$

repeat

if $i > mid$ then for $k \leftarrow j$ to $high$ do // 处理剩余元素//

$B(h) \leftarrow A(k);\ h \leftarrow h+1$

repeat

else for $k \leftarrow i$ to mid do

$B(h) \leftarrow A(k);\ h \leftarrow h+1$

repeat

endif

for $k \leftarrow low$ to $high$ do //把已归并的集合复制到A//

$A(k) \leftarrow B(k)$

repeat

end MERGE

注：n 个元素最初存放在 $A(1:\ n)$，调用 call MERGESORT(A, 1, n) 将使这 n 个元素排好序且仍存于 $A(1:\ n)$ 中.

下面来分析其时间复杂度。从根本上说，关键在于分析归并所需的比较次数。

归并两个已排好序的序列 $A(1:\ n)$ 和 $B(1:\ m)$，当 m 远远小于 n 时，用二分插入法较优，仅考查 $m=1$ 时，所需比较的最大次数为 $\lfloor \log(n+1) \rfloor$；但当 m 与 n 大小相差不多时，所需比较的最大次数差不多为 $m+n-1$, 有下面定理为证。

定理 2.7 任何以比较为基础的算法归并排好序的两个序列 $A(1:\ n)$ 与 $B(1:\ n)$ 在最坏情况下所需的比较次数不少于 $2n-1$.

证明：在算法 2.10 中，归并两个已排好序的序列 $A(1:\ n)$ 与

$B(1:\ m)$最多需要的比较次数是$m+n-1$. 下面来说明当$m=n$时，这个上界是任何归并算法在最坏情况下都能达到的。我们来考查满足

$$B(1)<A(1)<B(2)<A(2)<\ \cdots\ <B(n)<A(n)$$

的两个待归并的序列，任何归并算法在归并此两个序列时必须执行下面$2n-1$个比较：

$B(1):A(1)$，　$A(1):B(2)$，$B(2):A(2)$，$\cdots$，

$A(n\text{-}1):B(n)$，$B(n):A(n)$

假如对某i，$B(i):A(i)$没有被执行，则该算法在输入这两个序列的情况下输出的结果与输入满足下式的两个序列时必然一样：

$$B(1)<A(1)<\cdots<B(i-1)<A(i-1)<A(i)<B(i)$$
$$<B(i+1)<A(i+1)<\cdots<B(n)<A(n),$$

显然必有一种输出是错误的，同样道理，若对某i，$A(i):B(i+1)$没被执行的话，则该算法在输入这两个序列的情况下输出的结果与输入满足下式的两个序列时必然一样：

$$B(1)<A(1)<\cdots<B(i-1)<A(i-1)<B(i)<B(i+1)$$
$$<A(i)<A(i+1)<\cdots<B(n)<A(n),$$

同样必有一种输出是错误的，得证。

注：从证明过程来看，应有：当$|m-n|=0,\ 1$时，任何以比较为基础的算法归并排好序的两个序列$A(1:\ n)$与$B(1:\ m)$在最坏情况下所需的比较次数不少于$m+n-1$. 定理说明归并算法 2.8 就比较次数而言在最坏情况下是最优的。

归并排序算法 2.7 的最坏时间复杂度（基本操作是元素间的比较）满足下列递归式：

$$\begin{cases} WT(1)=0, \\ WT(n)=WT(\lfloor n/2 \rfloor)+WT(\lceil n/2 \rceil)+n-1,\ n\geq 2 \end{cases} \tag{2.46}$$

由 **master** 定理(情形(2))立即可知道, $WT(n)=\Theta(n\log n)$, 于是我们得到了一种最坏时间复杂度为$\Theta(n\log n)$的排序算法。关于归并排序算法平均时间复杂度的分析，我们宁愿稍微往后放一放，由一个更为一般的结论来涵盖。

事实上，对归并排序在最坏情况下的比较次数作更为精确的估计是一件有趣的事。

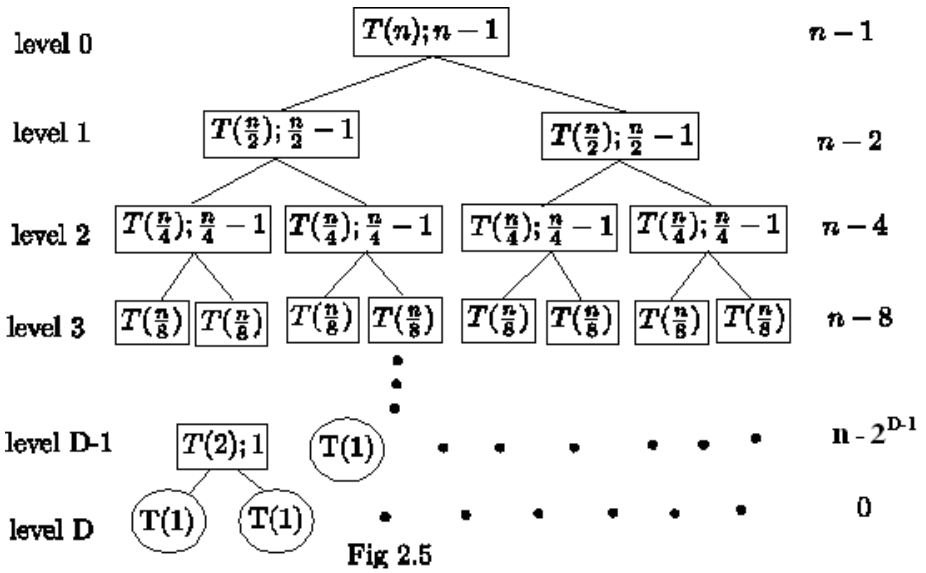

Fig 2.5

我们来考查归并排序形成的递归树（如 Fig 2.5 所示）。 注意到树的第d ($0\leq d\leq D-2$, 树的最深级$D=\lceil \log n \rceil$)级只含内点(这儿，内点指还需要继续分割的点)， 共有2^d个, 归并所花费的比较数总共是$n-2^d$; 树共有n个外点(指不需要分割的点，或者说，表示

对 1 个元素进行排序的问题)，外点处的花费是0. 外点只可能在第D级或第$D-1$级上，不管具体情况如何，第$D-1$级的结点(包括内结点和外结点)数是2^{D-1}，归并所花费的比较数仍是$n-2^{D-1}$，第D级只含外点，花费比较数是0。

基于以上事实，我们有

$$WT(n)=\sum_{d=0}^{D-1}(n-2^d)=nD-2^D+1. \tag{2.47}$$

我们再作进一步估计，令$\alpha_n=2^D/n$, 则

$$D=\log n+\log\alpha_n, \tag{2.48}$$

将(2.48)代入(2.47)得

$$WT(n)=n\log\ n-(\alpha_n-\log\alpha_n)n+1.$$

注意到

$$1\le\alpha_n<2,$$

从而

$$(1+\ln(\ln 2))/\ln 2\le\alpha_n-\log\alpha_n\le 1 \tag{2.49}$$

$$((1+\ln(\ln 2))/\ln 2\approx 0.914)$$

（注：计算函数$f(x)=x-\log x$，$1\le x<2$的值域）

最后我们得到：

$$n\log\ n-n+1\le WT(n)\le n\log\ n-0.914n+1. \tag{2.50}$$

4.3 快速排序(Quicksort)

快速排序是较早发现的分治排序算法，首先由霍尔(C.A.R.Hoare)发表于 1962 年。快速排序也是实际运算最快的排

序算法。

快速排序的基本思路如下：取待排序的 n 个元素 $A(low:high)$ 中的某个元素 v（譬如取 $v=A(high)$），确定 v 在这 n 个元素升序排列应处的位置 i，且重排 $A(low:high)$ 中元素使得 $A(low,i-1)$ 中的元素都不大于 v，$A(i)=v$，$A(i+1:high)$ 中的元素都不小于 v。然后对 $A(low,i-1)$ 和 $A(i+1:high)$ 分别进行快速排序。

算法 2.11 快速排序

Procedure Quicksort(low, $high$, A)

// $A(low:\ high)$ 是一个全程数组，它含有 $high-low+1\geq 0$ 个//

//待排序的元素。//

Integer i, j; Local v

If ($high>low$) then

{

$v\leftarrow A(high)$; $i\leftarrow low-1$

for $j\leftarrow low$ to $high-1$ do

if $A(j)\leq v$ then

$i\leftarrow i+1$;

if $i\neq j$ then $A(i)\leftrightarrow A(j)$ endif

endif

repeat

$i\leftarrow i+1$; $A(i)\leftrightarrow A(high)$

Quicksort(low, $i-1$, A); Quicksort($i+1$, $high$, A)

}

endif

end Quicksort

分析：同样以元素比较为基本操作，快速排序的最坏时间复杂度为$\Theta(n^2)$（这发生在$A(1:n)$中元素递增或递减排列时），平均时间复杂度为$\Theta(n\log n)$。事实上，设快速排序的平均时间复杂度（假定这些元素两两不同）为$T(n)$，则有下列递归式：

$$\begin{cases} T(1)=0 \\ T(n)=n-1+(2/n)\sum_{i=1}^{n-1}T(i) \end{cases}$$

与二元树插入排序算法的的平均时间复杂度$T(n)$满足的递推关系完全相同。

4.4 比较排序算法时间复杂度下界分析

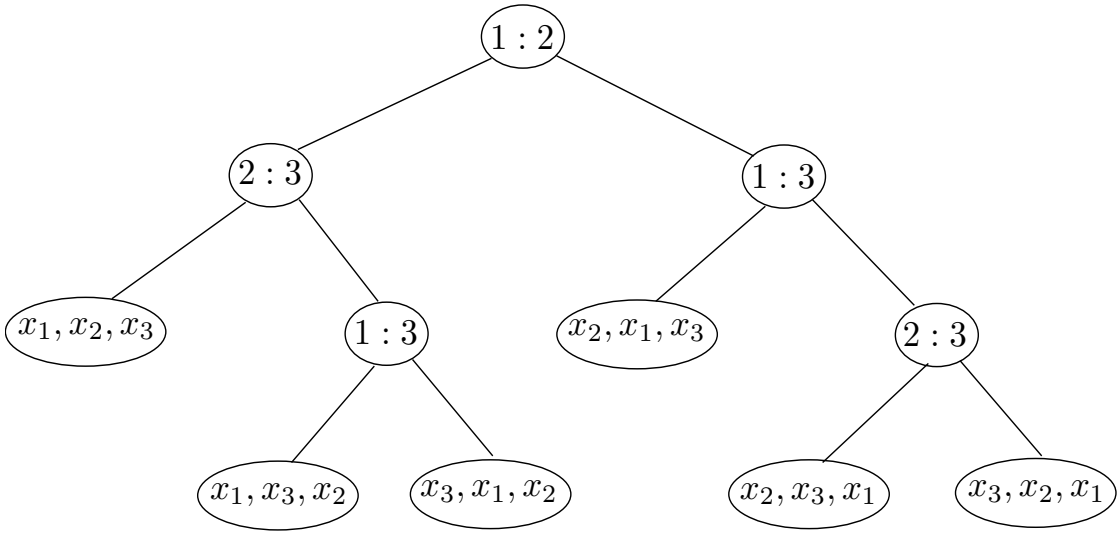

Fig 2.4 Decision tree for a sorting algorithm,n=3

任何以比较为基础的排序算法A对n个元素进行排序的情况都

可以用一个二元比较树(或二元决策树)来表示。图 2.4 表示某算法对 3 个元素进行排序的情况。该树的每个内结点表示一次比较操作，比较的结果总是将算法导向两个不同的状态之一；而该树的每个外结点表示一种可能的排序结果。所以该树至少有$n!$个外结点(假设n个元素两两不同)。一个具体的输入所需化费的比较数恰是表示所对应输出的外结点到根的距离。因此，算法A的最坏情况比较数是该树的高度，而平均比较数是该树所有外结点的级数的平均值。

任何二元树的高度h与外结点数目L有如下关系：$L \le 2^h$, 等价地, $h \ge \lceil \log L \rceil$. 所以二元决策树的高度$h \ge \lceil \log(n!) \rceil$,

$$\log(n!) = \sum_{i=1}^{n} \log i = \sum_{i=2}^{n} \log i > \sum_{i=2}^{n} \int_{i-1}^{i} \log x \, dx$$

$$= \int_1^n \log x \, dx = \log e \int_1^n \ln x \, dx = \log e \cdot (x \ln x - x) \Big|_1^n$$

$$= \log e(n \ln n - n + 1) = n \log n - n \log e + \log e$$

$$> n \log n - 1.443n + 1$$

另外，在所有含相同个数外结点的二元树中，以均衡二元树的外结点的平均级数为最小。均衡二元树的外结点的级数是h或$h-1$.

综上所述，任何比较排序算法的最坏情况比较数不小于$\lceil n \log n - 1.443n \rceil + 1$, 平均情况比较数不小于$n \log n - 1.443n$.

思考：以比较为基础判断n（$n \ge 2$）个元素中是否有相同的元素（对两元素a,b进行比较的输出为$a < b$或$a = b$或$a > b$）的任何算法的最坏时间复杂度为$\Omega(n \log n)$。

5. 选择问题与敌手对策方法

本节研究几个可统称为选择的问题，找n个元素(可比较大小)的中间元，就是其中一个著名的例子。我们除了设计解决这些问题的高效算法外，还将探讨这些问题的下界估计。我们通过引入一种被广泛应用的称为敌手对策的方法来做这件事。

5.1 选择问题

假设E是一个含有n个可比较大小元素的集合，k是一个正整数，$1 \le k \le n$, 选择问题就是寻找E中第k小元素的问题。我们约定可作的操作是：两元素的比较；元素拷贝或移动。而我们通常不太关心元素的移动。

除了找最大或最小元外，另一个常见的选择问题是找中间元，即在E中找第$\lceil n/2 \rceil$小元素。中间元在很多以数据表示的事件中是有典型意义的，譬如说，要说明某个国家或某个行业人员的收入，房屋的价格，大学录取分数线，这些涉及处理数据的事情，往往归结为求平均数或中间数的问题。平均数可在$\Theta(n)$时间内容易求得，那么如何有效的来计算中间元呢？

当然，所有的选择问题实例可通过对E排序来解决：$E(k)$正是第k小元素。排序需要$\Theta(n\log n)$数量的比较操作，而我们已经注意到对某些k (如，$k=1$, $k=n$)，选择问题可在线性时间内解决。直观上，找中间元应是最困难的选择问题。那么，我们能在线性时间内找出中间元吗？或者，我们能确定找中间元的时间复杂度的下界是线性甚至是$\Theta(n\log n)$吗？我们将在这一节回答这些问题，并给出

求解一般选择问题的算法框架。

5.2 下界估计

到目前为止，我们是利用决策树作为主要手段来对以比较为基本操作的问题进行下界估计的。在表示解决问题的任何算法的决策树中，每个内点表示算法可能执行的一次比较，而每个外结点表示一种可能的输出(在检索算法的情形下，每个内点也同时表示一种可能的输出)。算法在最坏情况下执行的比较次数正是决策树的高度(或深度)，该高度至少为$\lceil \log\ L \rceil$,其中，L是决策树的叶子（外结点）数目。

在检索问题中，决策树得到的下界估计是$\lceil \log(n+1) \rceil$, 能被二分检索算法达到，这当然是最理想的；而在排序问题中，决策树得到的下界估计是$\lceil \log(n!) \rceil$, 归并排序算法的复杂度几乎可以达到此界， 也相当令人满意。 但企图用决策树方法对选择问题作下界估计却不能奏效。

任何选择问题的决策树至少有n片叶子(作为输出)，因为n个元素的任何一个都可能是第k小元素，所以树的高度(即算法最坏情况下执行的比较数)至少是$\lceil \log n \rceil$, 这不是一个好的下界估计。我们知道，最简单的选择问题（找最大元或最小元）在最坏情况下至少需要$n-1$次比较。问题出在哪儿呢？在找最大元的决策树中，可能由多片叶子来表示同一种输出。 但我们没有一种简易的方法一般性地确定任何一种输出究竟由多少片叶子来表示。

为此，我们采用一种称之为敌手对策(Adversary Argument)的

方法，来取代决策树方法，对选择问题进行下界估计。

5.3 敌手对策

什么叫敌手对策呢？比方说，以警察捉小偷类比算法求解问题：把算法比作警察，输入比作小偷，算法对输入给出正确的输出比作警察捉住小偷。要对算法在最坏输入下的时间耗费作一个好的下界估计，必须把最坏的输入设想成一个狡猾的小偷，只要有机可乘，总是能逃脱警察的追捕，即让算法的每一步执行得到尽量差的结果。这就是敌手对策。再看一个更具体一些的例子，假定你与一个朋友做猜日期的游戏：你有权随意决定一天(365 天中)让朋友通过提出“是否问题”来猜，随意的意思是你不必事先确定这个日子，而视朋友的实际发问而定，尽量不让他猜着，直到被逼出一个日子为止。譬如，朋友问：“这一天是在冬天吗？”你当然回答：“不是!”因为不是冬天的日子更多。再如，朋友问：“这一天所在的月份的第一个字母（英文）是在字母表的前一半吗？”你应回答：“是!”，在此例中，你扮演的正是敌手的角色。

现在假定我们有一个以比较为基本操作的有效算法，设想存在一个敌手在背后操纵着它的输入，致使算法执行每次比较时，由敌手来决定比较的结果，敌手选择的结果总是使算法的进展缓慢，而敌手必须遵循的唯一规则是相容性：必存在一种输入，使算法在该输入下的运行，与敌手选择的所有结果一致。如果敌手迫使算法必须至少执行 $f(n)$ 次比较，则 $f(n)$ 就是算法在最坏情况下时间复杂度的一个下界估计。敌手的对策越高，得到的估计越好。

换一个角度，反过来看，“防御敌手对策”是有效求解基于比较操作的问题的好方法。就是说，设计算法时，尽量使得每次比较的结果相对均衡(两种结果的期望值一样)，以至敌手挑选哪一种，都区别不大。关于这一点，我们稍后再详细讨论，当下，我们的兴趣还是在用敌手对策方法估计下界。

我们想对一个问题的复杂度估计下界，而不是针对一个特殊的算法。所以，当我们采用敌手对策时，考虑的是某个算法类中的任意一个算法，就象我们应用决策树那样。为得到好的估计，我们必须请出强大的敌手来挫折任何算法。

竞赛术语：本节我们要给出选择问题的算法，以及一些情形下估计下界的敌手对策。在很多时候，我们采用竞赛术语，来表示比较结果。两比较元中的较大者称为胜者(winner),而另外一个称为败者(loser).

5.4 找最大与最小元问题的再讨论

本小节中，我们用 max 和 min 分别表示n个元素中的最大与最小元。

我们已经知道，单找 max 或 min 用$n-1$次比较就可以了，这样找完 max 后再找 min 需$n-1=n-1+(n-1-1)=2n-3$次比较。但这样做不是最优的方法，尽管$n-1$次比较对单找 max 或 min 是最优的，因为同时找二者的话，能使某些比较结果为二者所共享，一种设想是先让n个元素捉对比较(总共$\lfloor n/2 \rfloor$次)，然后在胜者(总共$\lceil n/2 \rceil$个元素)中找 max，在败者(也是总共$\lceil n/2 \rceil$个元素)中找 min,

由此设想得到的算法耗费的比较数为

$$\lfloor n/2 \rfloor + 2(\lceil n/2 \rceil - 1) = \lceil 3n/2 \rceil - 2.$$

下面用敌手对策来证明该算法是最优的。

定理 2.8 任何以比较为基础的找n元中 max 和 min 的算法在最坏情况下至少需$\lceil 3n/2 \rceil - 2$次比较。

所有比较结果列表

比较类型	结果	新增加信息单位
$N:N$	$N<N$	2
$N:W$	$N<W$	1
$N:L$	$N>L$	1
$N:WL$	$N>WL$	1
$W:W$	$W>W$	1
$W:L$	$W>L$	0
$W:WL$	$W>WL$	0
$L:L$	$L>L$	1
$L:WL$	$L<WL$	0
$WL:WL$	$WL<WL$	0

注：N表示没参加过比较的元素，W表示没输过的元素，LW表示没赢过的元素，WL表示赢过也输过的元素。

表 2.1

证明：为得此下界，我们仅考虑输入为n个两两不同元素的情形就够了。 为确定某元素x是 max 和某元素y是 min, 算法必须要知道异

于x的元素输过比较，异于y的元素赢过比较. 如果我们把某元素能赢或会输都记为一个信息单位，则算法必须得到$2(n-1)$个信息单位才能确保结果正确。

我们的敌手对策是让算法在每一次比较中得到尽量少的新信息单位。具体说来（如表 2.1 所示），就是如果比较是发生在从未参加过比较的两元之间，则一次比较可得到两个新信息单位，其他比较每发生一次至多允许得到一个新信息单位. 前一种比较的个数记为s, 后一种比较的个数记为t, 则总比较数为$s+t$, 且有：

$$s \le \lfloor n/2 \rfloor, \quad 2s+t \ge 2(n-1),$$

由此得：

$$s+t = 2s+t-s \ \ge \ 2n-2-\lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2 。$$

5.5 找第二大元素

我们当然可以找到n个元素中的 max 后, 再在剩余的$n-1$个元素中找 max 就得到 second-largest. 但会不会还有更有效的方法？进一步说，我们能证明某种方法是最优的吗？本小节回答这些问题。

因为要确定某元x是 second-largest, 必得知道x小于某元y, 而又大于其它$n-2$个元素。显然y必是最大元 max。所以在算法中，第一步先找 max 是个合理的设想。只是在找 max 的步骤中，希望提供尽可能多的信息使接下来找 second-largest 元花费的比较数尽量少。我们注意到， 在找最大元 max 的过程中，max 以外的其他$n-1$个元素必输过，若元素z输过非最大元 max 的某个元素， 则z

不可能是 second-largest。所以，我们仅在输过 max 的元素中找 second-largest 就够了。那么，如何在第一步找 max 的过程中，使与 max 比较过的元素数最少呢？我们采用锦标赛方法(Tournament Method)可做到这一点.

锦标赛方式：第一轮：所有元素分对比较；第二轮：第一轮中赢的元素分对比较；第三轮：第二轮中赢的元素分对比较；……, 直到决出最大元 max. 锦标赛过程可用一个二元树来表示，这是一个从底(bottom)到顶(top)的方式：最底层每片叶子包含一个元素(n片叶子)，配对的元素的父亲是其中的胜者(winner), 而最顶部的根包含的是 max. 在此锦标赛过程中，除 max 外，其它元素输过一次且仅一次，那么有多少元素输给 max 呢？如果$n=2^k$, 则恰有$\log n$个，一般地，该数目最坏为$\lceil \log n \rceil$ (注意到$\lceil \log \lceil n/2 \rceil \rceil + 1 = \lceil \log n \rceil$, 可用数学归纳法证明)，所以在第二步中，至多需$\lceil \log n \rceil - 1$次比较即可找到 second-largest, 总共花费

$$\lceil \log n \rceil - 1 + n - 1 = n + \lceil \log n \rceil - 2$$

次比较，下面我们来证明锦标赛方法是最优的。

定理 2.9 任何以比较为基础的找n个元素中 second-largest 元的算法在最坏情况下至少需$n + \lceil \log n \rceil - 2$次比较。

证明：我们用一个无向图来表示算法执行过程中所发生的比较：n个元素作为n个结点，若某两个元素作过比较，则在对应的两点之间连一线。如此得到的图必是连通图，而且删掉 max 点后，图仍是连通的。设与 max 相邻的点有t个，即 max 参加过t次比较，则算法至

少执行了$n-2+t$次比较。下面来说明在最坏情况下总有

$$t \geq \lceil \log n \rceil .$$

我们仍考虑n个元素各不相同，敌手给每个元素x一个权$w(x)$，起始所有元素的权都为1，敌手要求算法将依据权来决定发生比较的二元x与y的大小：

$$w(x) \geq w(y) \Rightarrow x > y$$

相应地，

$$w(x) \leftarrow w(x) + w(y),\ w(y) \leftarrow 0$$

任何时候元素的权和都是n，算法执行完毕时，max 元的权必为n，设 max 参加k次比较后的权为a_k，则

$$n = a_t \leq 2a_{t-1} \leq 2^2 a_{t-2} \leq \cdots \leq 2^t a_0 = 2^t$$

因此，

$$t \geq \log n$$

即

$$t \geq \lceil \log n \rceil$$

5.6 找中间元问题的一个下界估计

假设E是一个含n个元素的集合，n是奇数. 我们将针对任何以比较为基础找中间元(即第$(n+1)/2$小元素)的算法必须进行的比较数建立一个下界估计. 因为是考虑最坏情况的下界问题，可不失一般性地假定这些元素两两不同.

我们断言，要确定中间元median，算法必须得到其它元与中

间元的相对大小关系，即，对任意其他元 x，算法必须确知 $x > \text{median}$ 或 $x < \text{median}$。具体说来，算法必须确定：有 $(n-1)/2$ 个元小于 median，有 $(n-1)/2$ 个元素大于 median. 算法是通过建立如图 2.5 所示的树状关系来确定中间元的。图中，每个结点表示一个元素，每条边表示一次比较的结果，边的上端点表示获胜的元素，下端点表示认输的元素。图 2.5 确定了 9 个元素的中间元。而图 2.6 所表示的树状结构则不能确定中间元。图中，除结点 x 之外，其他结点均可能是中间元。

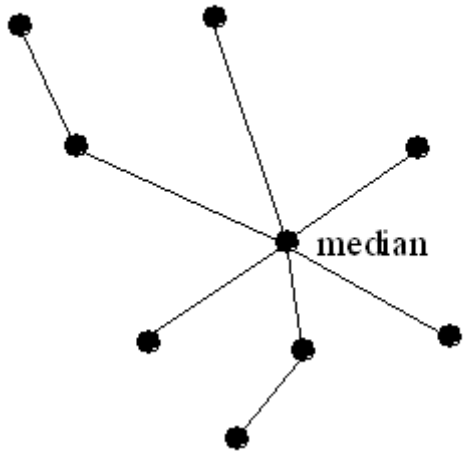

图 2.5

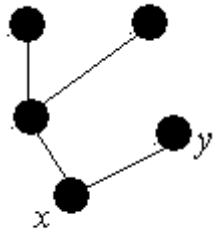

图 2.6

图 2.5 中的每个比较都是所谓的定性比较(crucial comparison)。

定义 2.3 在找中间元的算法执行过程中，一个涉及 x（x 最终未被算法确定为 median）的比较称为关于 x 的定性比较，如果该比较是第一次如下情形的比较：$x \geq y$ 且（算法执行的过程最终指明）$y \geq \text{median}$（此种情形下，x 被定性为 $x \geq \text{median}$）；或者，$x \leq y$ 且（算法执行的过程最终指明）$y \leq \text{median}$（此种情形下，x 被定性为 $x \leq \text{median}$）。

注：定义中，当x与y发生比较的时候，并不要求y与median的大小关系已经确定。

下面的讨论限于作为输入的n个元素两两不同。

任何找中间元的算法，如果其执行过程中，没有发生关于某个非中间元的定性比较，则算法不可能正确。细致说来，这是因为，假如算法在某次执行过程中，有一个元素x没有执行过关于自身的定性比较，换句话说，与x比较过的元素，要么没有定性；要么定性为大于median，同时在与x比较中获胜，或定性为小于median，同时在与x比较中认输。如此以来，则如下改变该次执行所对应的输入：(a). 若输入中，$x > \mathrm{median}$，则可改为$x < \mathrm{median}$，且能保持x与其他比较过的元素的相对大小；(b) 若输入中，$x < \mathrm{median}$，则可改为$x > \mathrm{median}$，且同样能保持x与其他比较过的元素的相对大小。由于算法的输出结果完全取决于所执行的比较，对输入作上述改变后，算法执行过程所发生的比较及其结果并无不同，所以算法给出的输出理应是一样的，显然必有一种情况下是错误的。所以为保证正确性，算法必然要执行$n-1$次定性比较。

图 2.6 中，x与y的比较是”无用的(useless)”比较，称之为非定性（noncrucial）比较。特别注意，在一般情形下，大于中间元的元素与小于中间元的元素之间的比较总是非定性比较。

我们下面将要说明，任何找中间元的算法除了必须执行$n-1$次定性比较外，可能执行的非定性比较数在最坏情况下至少为$(n-1)/2$。

定理 2.10 任何以比较为基础的找n（n为奇数）个元素中间元的算法在最坏情况下至少需要$3(n-1)/2$次比较。

证明：只需要说明任何算法执行的非定性比较数在最坏情况下至少为$(n-1)/2$。

我们确定敌手对策的原则是：迫使算法执行尽可能多的非定性比较。这儿给出一个较为简单的方案。此方案是通过控制这n个元素的取值而实现的。具体做法是：在算法执行过程中，给这n个元素分别赋值。此处，给某元素x赋值的意思，不是给它一个具体的值，而是赋予x三种状态之一：大于中间元(记此状态为L)；小于中间元(记此状态为L)，是中间元(记此状态为E)。x未被赋值时的状态记为N。任何元素x第一次参加比较时被赋值。 敌手赋值的规则如表 2.2 所示。

要注意，赋值为L的元素数恰为$(n-1)/2$，赋值为S的元素数也恰为$(n-1)/2$。执行敌手赋值规则不能违反此限制。所以，当已经有$(n-1)/2$个L元或$(n-1)/2$个S元的情况下，其余赋值不必继续遵循敌手赋值规则。而最后一个N元素被赋值为中间元。

比较类型	赋值结果
$N:N$	一个被赋值L，另一个被赋值S
$N:L$	被赋值S
$N:S$	被赋值L

表 2.2

表 2.2 涉及的比较都属于非定性比较。任何算法在此敌手对策下

需要执行多少次这样的非定性比较呢？在敌手赋值规则失效之前，任何一次赋值比较至多产生一个S元，至多产生一个L元。因此当产生了$(n-1)/2$个L元或$(n-1)/2$个S元的时候，算法被迫执行了至少$(n-1)/2$非定性比较。

注 1：证明中描述的敌手对策不能保证会发生多于$(n-1)/2$次非定性比较，这是因为算法可以开始先进行$(n-1)/2$次的$N:N$型比较。

注 2: 关于找中间元问题的下界, 目前得到的最好结果是比$2n$稍大一些, 与我们所知最好算法的比较数还有一点小差距。

5.7 防御敌手对策的设计方法

有一类操作譬如元素比较等，用来探求输入元素所含的信息（譬如顺序信息，相对大小信息等）。凭借这类基本操作所获得的信息来决定输出的算法（譬如排序算法， 检索算法，选择算法等），总是希望算法执行过程中发生的每次操作都得到尽可能多的信息。可惜并不能尽如人意，因为总有敌手对策在作对，而使操作往往呈现你不希望发生的结果。为了提高算法的质量，防御敌手对策的设计方法是一种很有效的技术。

防御敌手对策的设计方法，简而言之，就是要使敌手对策无可乘之机。具体来说，要使算法涉及的每个基本操作，不管结果如何，都能得到几乎一样多的信息。

从前面关于一些典型算法的讨论中可以发现，一个好算法需要引入某些平衡的想法。譬如用决策树来表示检索算法或排序算法执行情

况时，最坏情况下的比较次数是决策树的高度。在问题规模确定的情况下，为使决策树高度最小，就要尽量使树均衡化。决策树的每个内点表示一次比较，每个叶子表示算法的一种输出（有时，每个内点也表示一种输出，譬如在检索算法的决策树中，就是这样）。内点的左右子树分别表示比较的两种可能结果分别导向的算法输出数目。均衡树要求这两个数目基本相同， 即相同或相差为 1。二分检索算法和归并排序算法对应的决策树正是均衡树。

在找最大最小元问题和找第二大元问题的算法设计中，使用的锦标赛方法，也是典型的防御敌手对策的设计方法。

5.8 一般选择问题算法

一般选择问题：找n个元素$E(1:n)$中的第k小元素。

对此问题，下面介绍分治法给出的一种最坏时间复杂度为$O(n)$的算法。基本步骤：当$n \le 5$时，通过排序直接求解；当$n > 5$时，

1. 将$E(1:n)$中的元素分成$n/5$组，每组5个元素，然后分别找每组的中间元，然后再找这些中间元中的中间元m^*。如图 2.7 所示，以上操作将$E(1:n)$中元素分成四个区域。其中，C中元素比m^*小，B中元素比m^*大。
2. 接着A，D中的元素分别与m^*比较。到此，$E(1:n)$就分成比m^*小的元素集合S和比m^*大的元素集合L。
3. 若$k \le |S|$，则问题归结为找S中的第k小元素；若$k = |S| + 1$，则m^*正为所求；若$k > |S| + 1$，则问题归结为找L中的第$k - |S| - 1$小元素。

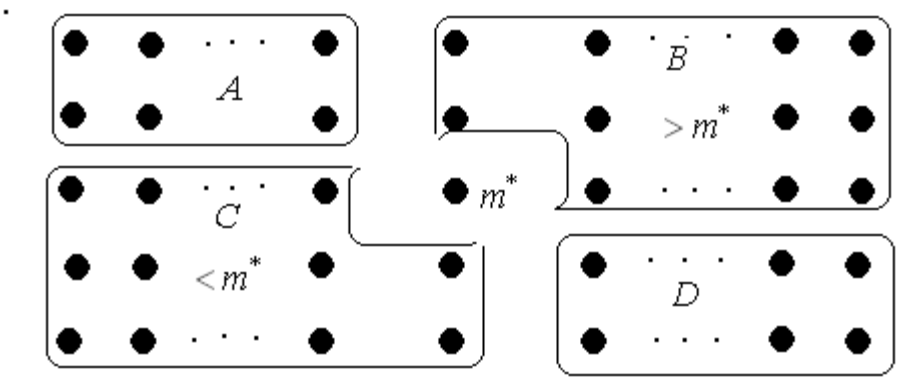

图 2.7

以上算法的最坏时间复杂度$W(n)$满足下面递归式：

$$W(n)\le cn+W(n/5)+W(7n/10)\quad（c\ge 1）\quad (2.51)$$

（更准确地，$W(n)\le cn+W(n/5)+\max\limits_{3n/10\le m\le 7n/10}W(m)$）

递归式(2.49)中，cn包含了在所有5元小组找中间元和A，D中的元素分别与m^*比较所需要的比较数；$W(n/5)$是指在小组中间元中找中间元m^*所需的最大比较数；$W(7n/10)$表示找S中的第k小元素或找L中的第$k-|S|-1$小元素所花费的最大比较数。

定理 2.11 $W(n)\le 20cn$。

证明：采用数学归纳法可证得。当$n\le 5$时，通过排序直接求解，显然有$W(n)\le 20cn$；当$n>5$时，由递归式（2.49）和归纳假设得

$$W(n)\le cn+W(n/5)+W(7n/10)\le cn+20c(n/5)+20c(7n/10)$$

$$=cn+4cn+14cn=19cn<20cn。$$

6. 平面上的最近点对和凸壳问题

本节考虑涉及平面上有限点集的两个著名问题. 这两个问题来自两个应用领域: 计算几何(computational geometry)和运筹学

(operations research)。

6.1 最近对问题

最近对问题是要找平面上n个点中的两个相距最近的点. 当然, 这些点可以是更高维空间的点。我们以笛卡儿坐标(x,y)的形式表示点P, 点$P_i=(x_i,y_i)$与点$P_j=(x_j,y_j)$之间的距离是指他们的欧氏距离

$$d(P_i,P_j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}.$$

容易想到的直接求解方法是，分别算出这n个点两两之间的距离，来找出最小距离的两点。

算法 2.10 直接求解

```
Procedure BruteForceClosestPoints(P,n,index1,index2)
//输入: 数组A(1:n)   n(n≥2)个点P1=(x1,y1)…, //
//Pn=(xn,yn); 输出:最近两点的下标 index1, index2 //
  dmin←∞
  for i←1 to n-1 do
   for j←i+1 to n do
    d←sqrt((xi-xj)^2+(yi-yj)^2)
    If d<dmin then dmin←d; index1←i; index2←j
    endif
   repeat
  repeat
End BruteForceClosestPoints
```

直接求解算法的时间复杂度是数据无关的，为$\Theta(n^2)$。效率太低，下面利用分治法可得到最坏时间复杂度为$\Theta(n\log n)$的算法。

基本想法：将这n个点分成数目相同的两组，将原问题转化为两个子问题，在子问题求解的基础上，得到原问题的解。具体步骤如下：

1．将这n个点按y坐标排成升序，置放在数组$Y(1:n)$中；

2．将数组$Y(1:n)$按x坐标以稳定排序算法排成升序，置放在数组$X(1:n)$中；

（前面2个步骤作为下面递归算法的预处理。）

3．当$n\le 3$时，直接求解。否则，将数组$X(1:n)$分割成$X(1:\lfloor n/2\rfloor)$和$X(\lfloor n/2\rfloor+1,n)$，记点$X(\lfloor n/2\rfloor)$的横坐标为$x_m$，则数组$X(1:\lfloor n/2\rfloor)$和$X(\lfloor n/2\rfloor+1,n)$的点分别处于垂直线$x=x_m$的左右两侧，如图2.8所示。相应地，将数组$Y(1:n)$分割成与这两个子数组包含相同元素的两个数组$Y(1:\lfloor n/2\rfloor)$和$Y(\lfloor n/2\rfloor+1,n)$。时间复杂度$O(n)$。

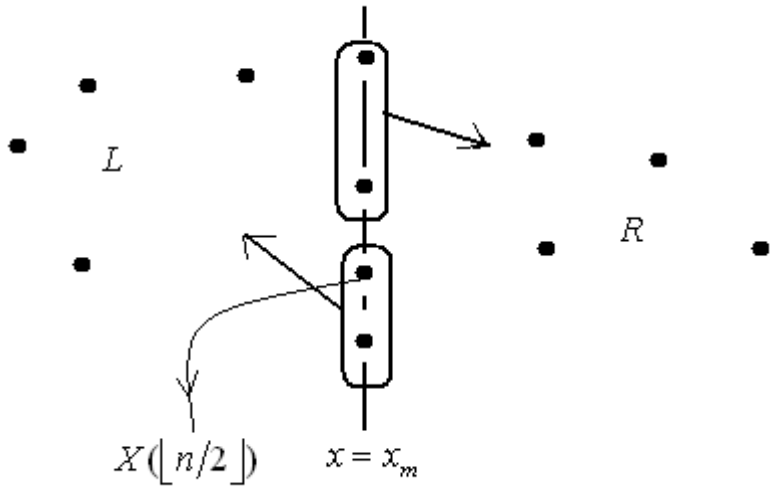

图2.8

4. 以$X(1:\lfloor n/2 \rfloor)$和$Y(1:\lfloor n/2 \rfloor)$作为输入，求解这$\lfloor n/2 \rfloor$个点中距离最近的两点，并求得其距离d_1；以$X(\lfloor n/2 \rfloor+1, n)$和$Y(\lfloor n/2 \rfloor+1, n)$作为输入，求解这$\lceil n/2 \rceil$个点中距离最近的两点，并求得其距离$d_2$；

5. 记$d=\max(d_1, d_2)$。为求原问题的解，只剩下来考察区域L中点与区域R中点之间的最小距离$\bar{d}$。而只有当$\bar{d}<d$时，这个考察

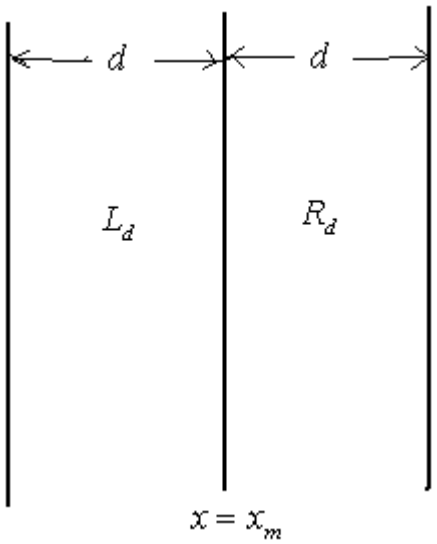

图 2.9

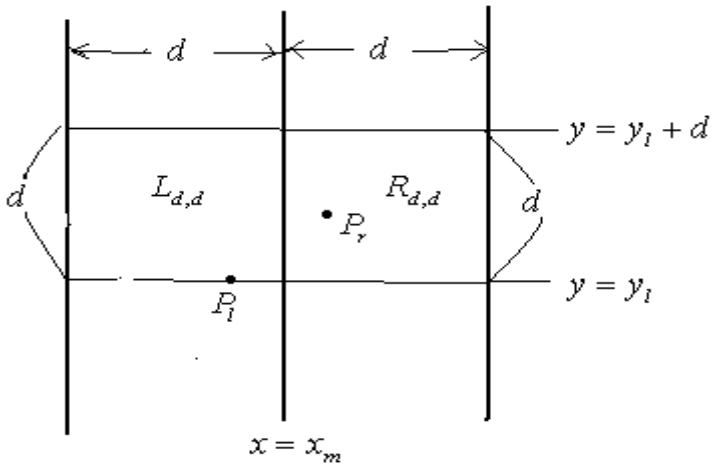

图 2.10

才有意义。因此只需要考察区域L_d(如图 2.9 所示)中点与区域R_d中

点之间的最小距离。从数组$Y(1:n)$中删去区域L_d和R_d以外的点得到数组$Y(1:n_1)$（$n_1 \le n$）。时间复杂度$O(n)$。

6．假设这个最小距离$\bar{d}$发生在如图 2.10 所示的P_l和P_r两点之间，即，$\bar{d} = d(P_l, P_r) < d$，且不妨假设$P_l$的纵坐标$y_l$不大于$P_r$的纵坐标。则$P_r$只可能处于两条水平线$y = y_l$和$y = y_l + d$之间，即$P_r$必在区域$R_{d,d}$中。不难看出区域$R_{d,d}$至多有$4$个点，同样区域$L_{d,d}$也至多有$4$个点。这说明要求得最小距离$\bar{d}$，只需对数组$Y(1:n_1)$中的每个点$P$，求$P$与（在数组中排在）它后面的 7 个点（实际上 5 个点足够了）之间的最小距离。时间复杂度$O(n)$。

设该算法（不考虑预处理）的最坏时间复杂度为$W(n)$，则

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + \Theta(n)$$

依据 **master** 定理得知：$W(n) = \Theta(n \log n)$. 由于预处理是两个排序过程，所以总时间复杂度依然是$\Theta(n \log n)$。可证明任何最近点对算法在最坏情况下的复杂度为$\Omega(n \log n)$。

6.2 凸壳(convex　hull)问题

先介绍一些基本概念和结论。

定义 2.4：平面上的点集S称为凸集，如果以S上的任何两点P和Q为端点的线段整个属于S。

图 2.11 是一些凸集，图 2.12 是一些非凸集。

定义 2.5：平面点集S的凸壳（记作$CH(S)$）是指包含S的最小凸集。

注：这儿“最小”的意思是：$CH(S)$是任何包含S的凸集的子

集。

凸集的凸壳当然是其本身，图 2.12 中非凸集的凸壳如图 2.13 所示。

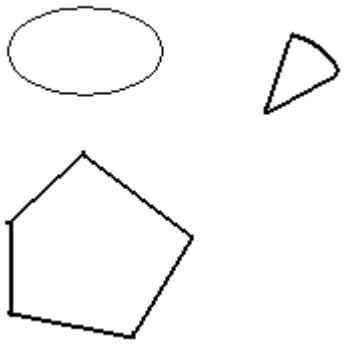

图 2.11

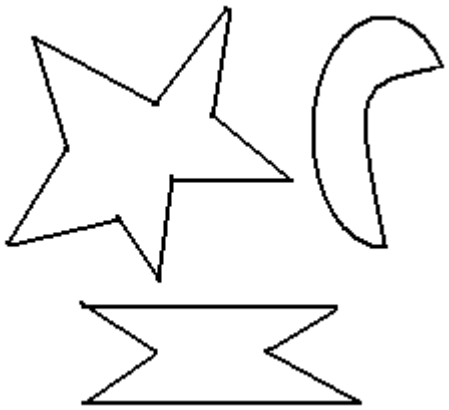

图 2.12

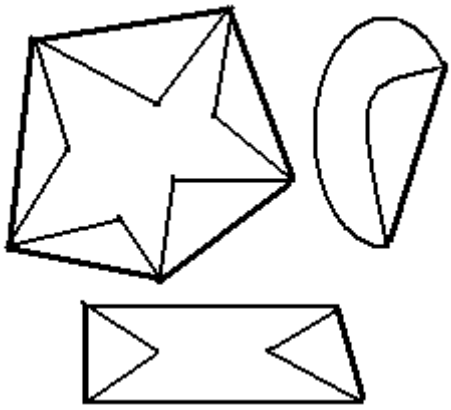

图 2.13

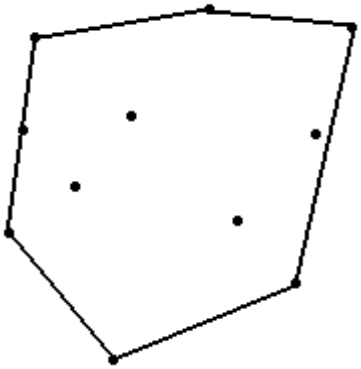

图 2.14

定理 2.11 设S是平面上不全在一条直线上的$n(n\geq 3)$个点组成的集合，则$CH(S)$是一个凸多边形，且凸多边形的顶点全是S中的点。如图 2.14 所示。

注：当S中的n个点全在一条直线上时，$CH(S)$是一条线段，且其两个端点是S中的点。

问题：给定平面上的n个点，求凸壳。

因为该问题的重要性，所以有多种算法，这儿介绍一种简单的分治算法。描述如下：

1. 分别找出横坐标最小和最大的点P_1，P_2，则P_1，P_2必是所求凸壳的顶点；
2. 将其他的点分割为直线P_1P_2上方的点和下方的点。分别在上方和下方的点中找与直线P_1P_2距离最远的点（如果存在的话）P_3，P_4（也必是所求凸壳的顶点）。
3. 继续分别找直线P_1P_3，P_3P_2上方的点，直至上方不再有点；
4. 继续分别找直线P_1P_4，P_4P_2下方的点，直至下方不再有点。

算法执行如图 2.15 所示。

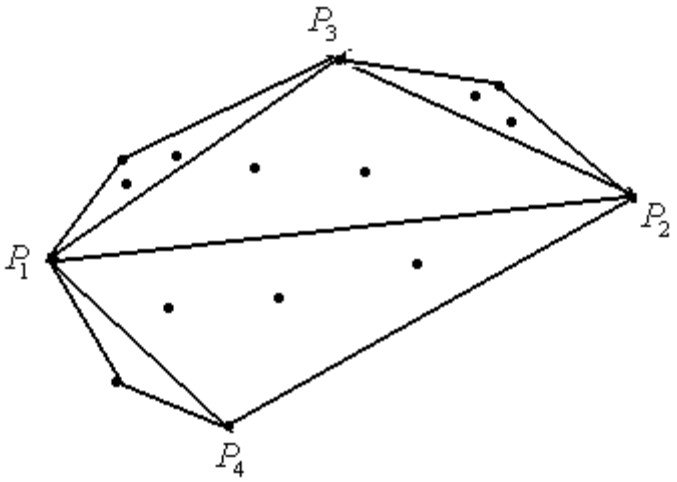

图 2.15

该算法最坏时间复杂度为$\Theta(n^2)$，平均时间复杂度为$\Theta(n\log n)$。

其他一些算法最坏时间复杂度可达到$\Theta(n\log n)$，但较为复杂。可证明任何凸壳算法在最坏情况下的复杂度为$\Omega(n\log n)$。

作业：

1. 一条直线上n个等间隔的格点上摆放n个电荷，电荷量分别为

$q_1,\cdots,q_n$，则第 j（$1\le j\le n$）个电荷受到的相互作用力为

$$F_j=\sum_{1\le i<j}\frac{cq_iq_j}{(j-i)^2}-\sum_{j<i\le n}\frac{cq_iq_j}{(j-i)^2}$$，（Coulomb's Law）

请设计时间复杂度为$O(n\log n)$的算法计算F_j。（参考书 20）

2. 求满足下列递归式的$T(n)$的渐近表示

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

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

(3) $T(n)=16T(n/4)+n^2$

(4) $T(n)=7T(n/3)+n^2$

(5) $T(n)=7T(n/2)+n^2$

(6) $T(n)=2T(n/4)+\sqrt{n}$

(7) $T(n)=T(2n/3)+T(n/3)+2n$

3. 假设n元序列

$$E(1),\ E(2),\ \cdots,\ E(n)$$

呈现单峰分布，即存在指标m $(1\le m\le n)$, 使得

$$E(1)<\cdots<E(m),\ E(m)>\cdots>E(n).$$

但我们不知道m的值. 问题：如何通过$\Theta(\log n)$次元素比较操作确定m？

4. 设$E_1(1:n)$和$E_2(1:n)$是增序排列的n元数组，要找出这$2n$个元素中的第n小元素。请设计时间复杂度为$O(\log n)$的算法求解该问题，并证明该问题的最坏时间复杂度为$\Omega(\log n)$。

5．在本节的假设条件下（x在表A中的n种情况的概率相同，x不

在表A中的$n+1$种情况的概率也相同），证明：在以比较为基础的所有检索算法中二分检索的平均比较次数最少。

6. 设T是结点数为$n=2^d-1$的完全二元树，其中d是T的最高级数。给T的每个结点v标定一个实数x_v，且这n个实数两两不同。某结点v称为T的一个局部最小点，如果与v有边相连的任何结点w标定的实数x_w都大于x_v。 请设计复杂度为$O(\log n)$的算法求T的一个局部最小点。

7. 给定$n\times n$（$n\geq 2$）的方形格点图G，每个格点(i,j)（$1\leq i\leq n$，$1\leq j\leq n$）的邻点为满足$|i-k|+|j-l|=1$的格点(k,l)。给每个格点v标定一个实数x_v，且这n^2个实数两两不同。某格点v称为G的一个局部最小点，如果v的任何邻点w标定的实数x_w都大于x_v。请设计复杂度为$O(n)$的算法求G的一个局部最小点。

8. 下列归并排序算法是数据无关的，请给出以比较为基本操作的时间复杂度。

procedure MERGESORT(A, low, $high$)

//$A(low:\ high)$是一个全程数组，它含有$high-low+1\geq 0$个//

//待排序的元素, 使用辅助数组$B(1:\ \lfloor (high-low)/2\rfloor+2)$和//

//$C(1:\lceil (high-low)/2\rceil+1)$进行归并//

integer low, $high$, mid

if ($low<high$) then

{

$mid\leftarrow\lfloor (low+high)/2\rfloor$ //分割点 //

call MERGESORT($A,\ low,\ mid$)

//子集合排序//

call MERGESORT($A,\ mid+1,\ high$)

//另一子集合排序//

for $i \leftarrow 1$ to $mid - low + 1$ do

$B(i) \leftarrow A(i + low - 1)$

repeat

for $j \leftarrow 1$ to $high - mid$ do

$C(j) \leftarrow A(j + mid)$

repeat

$B(mid - low + 2) \leftarrow \max;\ C(high - mid + 1) \leftarrow \max$

$i \leftarrow 1;\ j \leftarrow 1$

for $k \leftarrow low$ to $high$ do

if $B(i) \le C(j)$ then $A(k) \leftarrow B(i);\ i \leftarrow i + 1$

else $A(k) \leftarrow C(j);\ j \leftarrow j + 1$

endif

repeat

}

endif

end MERGESORT

设$t(n)$是上述算法对n个元素排序所花费的比较次数，则有

$$\begin{cases} t(1)=0 \\ t(n)=t(\lceil n/2 \rceil)+t(\lfloor n/2 \rfloor)+n,\ n\geq 2 \end{cases}$$

请证明：$t(n)=n\lceil \log n \rceil + n - 2^{\lceil \log n \rceil}$.

9．请设计一个算法，计算n个元素（可比较大小）组成的排列的反序数，要求算法的时间复杂度为$O(n\log n)$。

10．考虑幂函数 $f(x)=x^n$ 的数值计算问题，其中x是正实数，n是正整数。请编制时间复杂度为$O(\log n)$的算法求解该问题。并证明以实数乘法为基本操作求解该问题的任何算法的时间复杂度$t(n)$都满足：$t(n)=\Omega(\log n)$。

11. 给定实数数组$A(1:n)$和实数x，求解是否存在元素$A(i)$，$A(j)$（$i\neq j$），使得$x=A(i)-A(j)$。请给出最坏时间复杂度为$O(n\log n)$的算法求解该问题。

12. 给定非递减实数数组$A(1:n)$，$B(1:m)$和实数x，求解是否存在元素$A(i)$，$B(j)$（$1\leq i\leq n,\quad 1\leq j\leq m$），使得$x=A(i)-B(j)$。请给出最坏时间复杂度为$O(n+m)$的算法求解该问题。

13．矩阵$A=(a_{i,j})_{m\times n}$中的元素取自某有序集，每行元素从左到右呈非递减排列。若将A中每列元素排序为自上而下非递减的，得到矩阵$\overline{A}$，证明：$\overline{A}$中每行元素依然从左到右呈非递减排列。

14. 利用魔鬼策略设计算法：

(a). 请设计算法求5个元素的中间元，要求在最坏情况下只用6次比较。

(b). 请设计算法给5个元素排序，要求在最坏情况下只用7次比较。(注： 注意到$\lceil \log(5!) \rceil = 7$，少于$7$次比较不能保证在任何情况下给$5$个元素排序）

15. 给定平面上的n条直线$L_i : y = a_i x + b_i (1 \le i \le n, \quad a_i \ne 0)$, 假定其中任三条直线不会相交于同一点。称某直线$L_j$在给定横坐标$x_0$处是最高的，若$L_j$上对应点的纵坐标大于其他直线上对应点的纵坐标，即$a_j x_0 + b_j > a_i x_0 + b_i$对$i \ne j$成立。称直线$L_j$是可见的，若$L_j$在某横坐标点处是最高的。请设计时间复杂度为$O(n \log n)$的算法，来给出这$n$条直线中所有的可见直线。