电子版课堂笔记之

课程：算法导论
时间：2006.3-2006.7

~~班~~级：计算机系 工程硕士
主讲：黄刘生教授

笔记：王海龙
整理：李春生

Ch 1. 引言

1.1 Alg（算法）

形式定义：

任何一个良定义的运算过程

input → 算法 → output

（计算步骤）

例 排序问题：

Input：n 个数序列 $\langle a_1, a_2, \cdots, a_n \rangle$

Output：找一个重新的排列 $\langle a_1', a_2', \cdots, a_n' \rangle$

$a_1' \le a_2' \le \cdots \le a_n'$

计算步骤：如何达到上述关系.

实例（Instance）：计算问题的解所需的所有输入.

正确性：若算法对每个输入实例，算法均能终止于正确的输出，则算法为正确的.

不正确算法：对某些实例不停机（终止）

虽然停机，但输出并非用户希望的.

1.2 算法分析.

目的：估计算法所需资源（时间，空间，带宽）

计算模型：

单处理机 RAM模型

涉及知识

时间分析

算法耗费时间 $\begin{cases} \text{输入实例的大小} \\ \text{实例的构成} \end{cases}$

运行时间

最坏情况时间分析 对任何输入实例，算法的最长运行时间

平均时间分析： 所有输入实例是等概率

若实例并非如此，可随机化

1.3 算法设计

增量法 插入排序

分治法

贪心法

动态规划

回溯法

分枝-界限

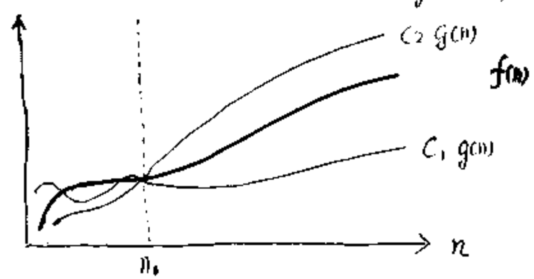
Ch 2 函数增长率

渐近时间

1. θ-记号

定义 Def 给定一个函数 $g(n)$，$\theta(g(n))$ 表示一个函数集合

$$\theta(g(n)) = \{f(n) \mid \exists \text{常数 } c_1, c_2, n_0 > 0, \text{使对所有 } n > n_0, \text{有 } 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\}.$$



$f(n) \in \theta(g(n)) \Rightarrow f(n) = \theta(g(n))$

↑算法时间 ↑算法时间量级

$f(n)$ 在一常数因子范围内等于 $g(n)$

$g(n)$ 是 $f(n)$ 的渐近上界和渐近的下界

$g(n)$ 是 $f(n)$ 的渐近的紧致界

注意(note)： θ 定义中 要求 $f(n)$ 和 $g(n)$ 是渐近非负的

记号 $\theta(1)$ 表示算法的运行时间与问题的规模无关

可写成 $\theta(n^0)$

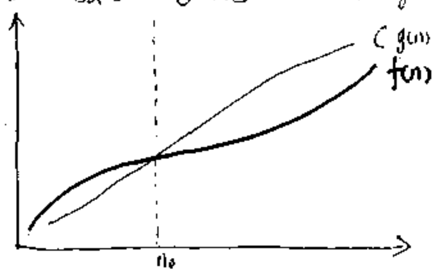
2. 渐近上界：O-记号：

定义：对给定的函数 $g(n)$，$O(g(n))$ 表示一个函数集合

$$O(g(n)) = \{f(n) \mid \exists 常数\ c, n_0 > 0，使得对所有\ n \ge n_0\ 有\ 0 \le f(n) \le c\,g(n)\}$$

即：在一个常数因子范围内，$g(n)$ 是 $f(n)$ 的渐近上界。

θ 记号 强于 O 记号： $\theta(g(n)) \subseteq O(g(n))$



$$\Downarrow$$

$$f(n) = \theta(g(n)) \Rightarrow f(n) = O(g(n))$$

注意：O：界定一算法的最坏时间

例：插入排序：

最好：线性阶

最坏：平方阶

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

$n = O(n^2)$　　$n \neq \theta(n^2)$

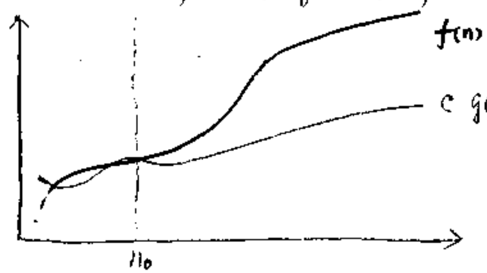
$n^2 = O(n^2)$　　$n^2 = \theta(n^2)$

3 Ω 记号（渐近下界）

定义：对给定的函数 $g(n)$，$\Omega(g(n))$ 表示一个函数集合

$$\Omega(g(n)) = \{f(n) \mid \exists 常数\ c, n_0 > 0，使得对所有\ n \ge n_0\ 有\ 0 \le c\,g(n) \le f(n)\}$$

Th 2.1 对任意函数 $f(n)$ 和 $g(n)$，$f(n) = \theta(g(n))$ 当且仅当 $f(n) = O(g(n))$ 和 $f(n) = \Omega(g(n))$



例：插入排序：
$$\begin{cases} T_1(n) = O(n^2) \\ T_2(n) = \Omega(n) \end{cases}$$

4. 方程中的渐近记号：

基于比较排序时间下界　$\lg n!$

$$\lg n! = n\lg n - 1.44n + O(\lg n)$$

5. 小o 记号 （渐近非紧致上界）

大O： $2n^2 = O(n^2)$ 　∵ $2n^2/n^2 \Rightarrow$ 常数2. 为紧致界.

$2n = O(n^2)$ 　$\left.\begin{array}{l} 2n/n^2 \to 0 \\ n^2/2n \to +\infty \end{array}\right\}$ 为非紧致界

定义：对给定的函数 $g(n)$ 　$o(g(n))$ 表示一个函数集合

$$o(g(n)) = \{ f(n) \mid \text{对}\ \forall\ \text{常数}\ c>0,\ \exists\ \text{常数}\ n_0>0,\ \text{对任何}\ n \ge n_0\ \text{有}\ 0 \le f(n) < c\,g(n) \}$$

例： $2n = o(n^2)$ 　∵ $2n/n^2 \to 0$

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

直观上　当 $n\to\infty$，$f(n)$ 相对于 $g(n)$ 可忽略.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

6. 小ω 记号 （渐近非紧致下界）

定义：对给定的函数 $g(n)$ 　$\omega(g(n))$ 表示一个函数集合

$$\omega(g(n)) = \{ f(n) \mid \text{对}\ \forall\ \text{常数}\ c>0,\ \exists\ \text{常数}\ n_0>0,\ \text{对任何}\ n \ge n_0\ \text{有}\ 0 \le c\,g(n) < f(n) \}$$

g 量级 < f 量级

例： $n^2/2 = \omega(n)$

$n^2/2 \neq \omega(n^2)$

$$f(n) = \omega(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

7. 函数间的比较：

假定 $f(n)$ 和 $g(n)$ 渐近非负.

1) 传递性：

$f(n) = \Theta(g(n))$ 并且 $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ 并且 $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ 并且 $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ 并且 $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$

$f(n) = \omega(g(n))$ 并且 $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

2) 自反性：

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

$f(n) = O(f(n))$

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

3) 对称性：

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

4) 转置对称性：

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

$f(n) = o(g(n))$ 当且仅当 $g(n) = \omega(f(n))$

利用上述4个性质，可将渐近比较类比于两实数间比较.

$f(n) = O(g(n)) \approx a \le b$ // "≈" 相似于 $f\text{-}a$, $g\text{-}b$

$f(n) = \Omega(g(n)) \approx a \ge b$

$f(n) = \Theta(g(n)) \approx a = b$

$f(n) = o(g(n)) \approx a < b$

$f(n) = \omega(g(n)) \approx a > b$

实数三歧性： 决定任何两实数是可比的

但任何两函数之间不一定能进行渐近比较.

$\exists f(n)$. $g(n)$ 可能

$f(n) = O(g(n))$ 不成立

$f(n) = \Omega(g(n))$ 也不成立

$\therefore f, g$ 不可比较

例：函数 n, 和 $n^{1+\sin n}$ 之间不可渐近比较.

$1 + \sin n \Rightarrow 0 \sim 2$

$n^{1+\sin n}$ 是在 $O(1)$ 到 $O(n^2)$ 之间波动.

ch 3. 求和

3.1 求和公式的性质

有限和 $\sum_{k=1}^{n} a_k$

约定：① $n=0$, 和式为0

② 当 n 不是整数，上限为 $\lfloor n/2 \rfloor$

当 x 不是整数，下限为 $\lfloor x \rfloor$

无限和 $\sum_{k=1}^{\infty} a_k$. 含义 $\lim_{n\to\infty} \sum_{k=1}^{n} a_k$.

1. 线性性质：

① $\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$ $(c \in R)$

② $\sum_{k=1}^{n} \Theta(f(k)) = \Theta(\sum_{k=1}^{n} f(k))$

2. 算术级数：

$\sum_{k=1}^{n} k = \Theta(n^2)$

3. 几何级数：

$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$. 当 $n \to \infty$ 且 $|x| < 1$ (无穷递减)

$\sum_{k=0}^{\infty} x^k = \frac{1}{x-1}$

4. 调和级数

$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$

5. 积分级数 或 微分级数

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ $(|x|<1)$

$\Downarrow$ (先微分再乘 x)

$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

6. 套迭级数

$$\sum_{k=1}^{n}(a_k-a_{k-1})=a_n-a_0$$

$$\sum_{k=0}^{n}(a_k-a_{k+1})=a_0-a_n$$

$$\left(\sum_{k=1}^{n-1}\frac{1}{k(k+1)}=\sum_{k=1}^{n-1}\left(\frac{1}{k}-\frac{1}{k+1}\right)=1-\frac{1}{n}\right.$$

7 积.

$$\prod_{k=1}^{n}a_k$$

当 $n=0$ 时，定义值为1.

$$\lg\left(\prod_{k=1}^{n}a_k\right)=\sum_{k=1}^{n}\lg a_k$$

3.2 和式的界

1. 数学归纳法

猜测和式数量级

例：证几何级数 $\sum_{k=0}^{n}3^k$ 为 $O(3^n)$.

证明：

① 归纳基础　$n=0$　$\sum_{k=0}^{0}3^k=1\leq c\cdot 3^0=c$

只要 $c\geq 1$，则归纳基础为正确的.

② 归纳假设：假设对 $n>c$ 成立.　$f(n)\leq c\,g(n)$

③ 归纳步骤.　对 $n+1$

$$\sum_{k=0}^{n+1}3^k=\sum_{k=0}^{n}3^k+3^{n+1}\leq c\cdot 3^n+3^{n+1}=\left(\frac{1}{3}+\frac{1}{c}\right)\cdot c\,3^{n+1}$$

$$\leq c\;3^{n+1}$$

只要 $\frac{1}{3}+\frac{1}{c}\leq 1$ 时 不等式成立，$\Rightarrow c\geq\frac{3}{2}$

$\therefore$ 只要 $c\geq\frac{3}{2}$. 则 $\sum_{k=0}^{n}3^k=O(3^n)$

注意(Note)：　小心使用渐近记号.

例 证 $\sum_{k=1}^{n}k=O(n)$

证明(pf)：　$\sum_{k=1}^{1}k=O(1)$

假设对 n 成立，则对 $n+1$

$$\sum_{k=1}^{n+1}k=\sum_{k=1}^{n}k+(n+1)=\underline{O(n)+(n+1)}\times$$

$$=O(n+1)$$

2. 对项限界：

① 最大/小项限界.

$$\sum_{k=1}^{n}k\leq\sum_{k=1}^{n}n=n^2=O(n^2)$$

一般情况. $\sum_{k=1}^{n}a_k\leq n\,a_{max}$

② 用几何级数限界：

假定 $\sum_{k=0}^{n}a_k$ 对所有 $\frac{a_{k+1}}{a_k}\leq r$，这里常数 r 满足 $0<r<1$，则

$\underline{a_k\leq a_0\cdot r^k}$　$(\because a_k\leq a_{k-1}\cdot r\leq a_{k-2}\cdot r^2\cdots\leq a_0\cdot r^k)$

∴ $\sum_{k=0}^{n} a_k \le \sum_{k=0}^{\infty} a_0 r^k$

$= a_0 \cdot \frac{1}{1-r} = \frac{a_0}{1-r}$

例 $\sum_{k=1}^{\infty} \frac{k}{3^k}$

$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3}\cdot\frac{k+1}{k} \le \frac{2}{3}$

∴ 当对于k≥1时，成立

∴ $\sum_{k=1}^{\infty} \frac{k}{3^k} \le \frac{1}{3} \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)^k = \frac{1}{3} \cdot \frac{1}{1-\frac{2}{3}} = 1$

注意(Note):

$\frac{a_{k+1}}{a_k} \le r < 1$ <满足要求，找到常数r>

例: $\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n\to\infty}\sum_{k=1}^{n} \frac{1}{k} = \lim_{n\to\infty}\Theta(\ln n) = \infty$

$\frac{a_{k+1}}{a_k} = \frac{1/(k+1)}{1/k} = \frac{k}{k+1} < 1$

3. 和式分解:

① 简单：一分为二

求 $\sum_{k=1}^{n} k$ 下界:

$\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=\frac{n}{2}+1}^{n} k \ge \sum_{k=1}^{n/2} 0 + \sum_{k=\frac{n}{2}+1}^{n} \frac{n}{2} \ge \left(\frac{n}{2}\right)^2$

$= \Omega(n^2)$

② 忽略和式初始几项:

$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k$ (k_0为常数)

$= \Theta(1) + \sum_{k=k_0}^{n} a_k$

例: $\sum_{k=0}^{\infty} \frac{k^2}{2^k}$

∵ $\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \le \frac{8}{9}$ (只对k≥3时成立)

∴ $\sum_{k=0}^{\infty} \frac{k^2}{2^k} = O(1) + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \le O(1) + \frac{9}{8}\sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1)$

③ 更复杂划分

$H_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \left(\frac{1}{2}+\frac{1}{3}\right) + \left(\frac{1}{4}+\cdots+\frac{1}{7}\right) + \left(\frac{1}{8}+\frac{1}{9}+\cdots+\frac{1}{15}\right) + \left(\frac{1}{2^{\lfloor \lg n\rfloor}}+\cdots+\frac{1}{n}\right)$

($i=0$ $\quad i=1$ $\quad i=2$ $\quad i=3$)

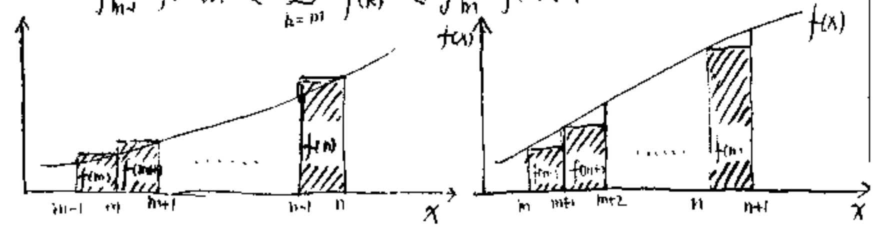
$\le \sum_{i=0}^{\lfloor \lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i+j}$

$\le \sum_{i=0}^{\lfloor \lg n\rfloor}\sum_{j=0}^{2^i-1}\frac{1}{2^i} \le \sum_{i=0}^{\lfloor \lg n\rfloor} 1 \le \lg n + 1$

4. 积分近似:

① 若f(k)单调递增，则

$\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx$



② $f(x)$ 为单调减：

$$\int_m^{n+1} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)\,dx$$

例：求 H_n 的紧致界

$\because f(k) = \frac{1}{k}$ 单调减

$$\sum_{k=1}^{n}\frac{1}{k} \ge \int_1^{n+1}\frac{dx}{x} = \ln(n+1) - \ln 1 = \ln(n+1)$$

$$\sum_{k=1}^{n}\frac{1}{k} = 1 + \sum_{k=2}^{n}\frac{1}{k} \le 1 + \int_1^{n}\frac{dx}{x} = \ln n + 1$$

$\therefore$ 和式紧致界 $\Theta(\ln n)$

Ch 4. 递归式和分治法

分治法思想

将一问题分解为与原问题相似，但规模更小的若干个子问题，递归地解这些子问题，然后将这些子问题的解组合起来，构成原问题的解。

在每层递归上有三个步骤：

① 分解（divide）：将问题划分为若干子问题

② 解子问题（conquer）：递归地解这些子问题

若子问题的 size（规模）足够小，则直接求解

③ 组合（combine）：将子问题的解组合成原问题的解

例1. $n! = \begin{cases} n\cdot(n-1)! & n>1 \\ 1 & n=1 \end{cases}$

例2：① 归并排序

② 快速排序

分治法的分析：

设 $T(n)$ 是 size 为 n 的执行时间

若 size 足够小，若 $n \le C$（常数），则直接求解的时间为 $\Theta(1)$

设分解的问题个数为 a，每个子问题大小为 $\frac{n}{b}$

则解各子问题的时间为 $aT(\frac{n}{b})$

设分解时间为 $D(n)$

组合时间为 $C(n)$

$$T(n)=\begin{cases}\theta(1) & \text{if } n\le c\\ aT(\frac{n}{b})+D(n)+C(n) & \text{if } n>c\end{cases}$$

$(b>1)$

一般地可忽略一些细节:

① 函数参数为整数:

$T(n/2) = T(\lfloor n/2\rfloor)$ / $T(\lceil n/2\rceil)$

② 边界条件可忽略,

当n较小: $T(n)=\theta(1)$

4.1 替换法.

① 猜测解: 确定常数c

② 将猜测的解代入递归关系式中.

例 $T(n)=2T(\lfloor n/2\rfloor)+n$ 上界.

解: 猜测 $T(n)=O(n\lg n)$

要证: $T(n)\le cn\lg n$ 对 $c>0$ 成立

假设它对 $\lfloor n/2\rfloor$ 成立. 即 $T(\lfloor n/2\rfloor)\le c\lfloor n/2\rfloor\lg(\lfloor n/2\rfloor)$

将其代入递归式中.

$T(n)\le 2c\lfloor n/2\rfloor\lg\lfloor n/2\rfloor+n$

$\le cn\lg(n/2)+n$ ※将"$\lfloor\ \rfloor$"去掉.

$= cn\lg n-cn+n$

$\le cn\lg n$ ※只要 $c\ge1$, 即 $-cn+n\le0$.

对边界亦成立

设 $T(0)=0$, $T(1)=1$,

$T(1)\le c\cdot\lg 1=0$ 不成立

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

$T(2)\le c\,2\lg 2=2c$

$4\le 2c$ // 只要 $c\ge2$

1. 做出好的猜测

① 与见过的解类似

例: $T(n)=2T(\lfloor n/2\rfloor+17)+n$.

则猜测为 $O(n\lg n)$.

② 先证比较宽松的上下界, 再减小猜测范围.

例: $T(n)=2T(n/2)+n$

下界 $\Omega(n)$

上界 $O(n^2)$, 子问题个数不会超过 $O(n)$.

推测 $\Rightarrow O(n\lg n)$.

No.

Date

2 细节

猜测的解中减去一个低次项

例: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

解为 $O(n)$ 要证 $T(n) \leq cn$

假设对 $\lfloor n/2 \rfloor$ 成立 则 $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor$

代入 $T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1$

$= c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + 1$

$= cn + 1$

从解中减去一个常数, 则猜测解为

$T(n) \leq cn - b$ // 常数 $b \geq 0$

代入 $T(n) \leq (c \cdot \lfloor n/2 \rfloor - b) + (c \cdot \lceil n/2 \rceil - b) + 1$

$= cn - 2b + 1$

$\leq cn - b$ // 只要 $b \geq 1$, $c > 0$

3. 避免陷阱: 不能直接使用渐近记号

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

猜解为 $T(n) \leq cn$

代入 $T(n) \leq 2c\lfloor n/2 \rfloor + n$

$\leq cn + n$

$\leq O(n)$ X // 错误!

No.

Date

4. 变量替换

例 $T(n) = 2T(\sqrt{n}) + \lg n$

令 $m = \lg n$ $n = 2^m$

原式改为 $T(2^m) = 2T(2^{m/2}) + m$ ①

令 $S(m) = T(2^m)$

①式为 $S(m) = 2S(m/2) + m$ //与前例相同

$\therefore S(m) = O(m\lg m)$

再回到T形式:

$T(n) = T(2^m) = S(m) = O(m\lg m)$

$= O(\lg n \lg\lg n)$

$\therefore T(n) = O(\lg n \lg\lg n)$

4.2 迭代法

① 展开

例 $T(n) = 3T(\lfloor n/4 \rfloor) + n$

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

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

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

第i项 $3^i \lfloor n/4^i \rfloor$

$3^i T(\lfloor n/4^i \rfloor)$ 边界为 $\lfloor n/4^i \rfloor \leq 1$

即 $i \geq \lg_4 n$

当 $i=\log_4 n$ 时，有 $T(1)=\theta(1)$

$$T(n) \le n + 3n/4 + 3^2 n/4^2 + \cdots + 3^{\log_4 n}\theta(1)$$
$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \theta(n^{\log_4 3})$$
$$= 4n + O(n) \quad // \text{小 } o$$
$$= O(n) \quad // \text{大 } O$$

关键　达到边界条件所需的迭代次数

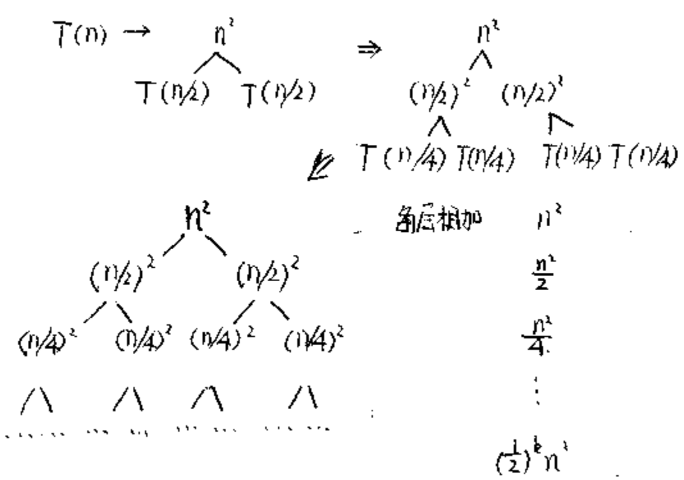
当 floor（地板函数 $\lfloor\ \rfloor$）和 ceiling（天花板函数 $\lceil\ \rceil$）出现时

结果为整数（可假定 n 是整数次幂）则可去掉符号。

② 递归树

展开过程直观化

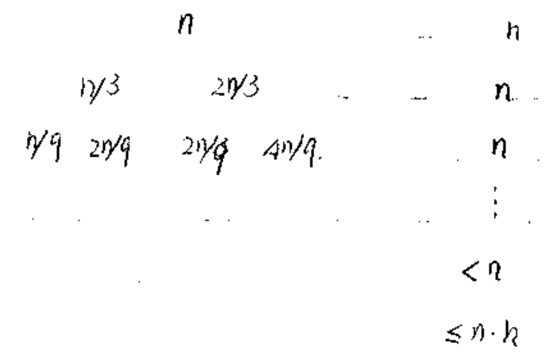
例　$T(n) = 2T(n/2) + n^2$



$n^2 \to \frac{1}{2}n^2 \to \frac{1}{4}n^2 \to \cdots \to (\frac{1}{2})^k n^2$

$(\frac{n}{2^k}) = 1$, $\quad k = \lg n$

例　$T(n) = T(n/3) + T(2n/3) + n$



树高 h

$n \to \frac{2}{3}n \to \frac{4}{9}n \to \cdots \to (\frac{2}{3})^k n$

令 $(\frac{2}{3})^k n = 1$

⇓

$n = (\frac{3}{2})^k$, $k = \log_{3/2} n$

$h = k+1 = O(\lg n)$

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

4.3　公式法（master 法）

设 $a\ge1$, $b>1$ 是整数，$f(n)$ 是函数

$T(n)$ 是定义在非负整数上的递归方程

$T(n) = aT(n/b) + f(n)$　这里 n/b 解释为 $\lceil n/b\rceil$ 或 $\lfloor n/b\rfloor$

则 $T(n)$ 的渐近界为：

1. 若 $f(n) = O(n^{\log_b a - \varepsilon})$ 对某常数 $\varepsilon>0$ 成立，

　则 $T(n) = \theta(n^{\log_b a})$

2. 若 $f(n) = \theta(n^{\log_b a})$，则 $T(n) = \theta(n^{\log_b a}\lg n)$

3. 若 $f(n) = \Omega(n^{\log_b a+\varepsilon})$ 对某常数 $\varepsilon>0$ 成立，

　且 $af(n/b) \le cf(n)$ 对某常数 $c<1$ 及足够大的 n 成立，

　则 $T(n) = \theta(f(n))$

No.

比较 $f(n)$ 和 $n^{\log_b a}$ 较大者决定 $T(n)$ 的解.

情况1: $n^{\log_b a} > f(n)$，则 $T(n) = \theta(n^{\log_b a})$

2. $n^{\log_b a} = f(n)$. 则 $T(n) = \theta(f(n) \cdot \lg n) = \theta(n^{\log_b a} \cdot \lg n)$

3: $n^{\log_b a} < f(n)$，则 $T(n) = \theta(f(n))$

在比较 $f(n)$ 和 $n^{\log_b a}$ 的大小时，一定要大于一个多项式因子 $n^{\varepsilon}(\varepsilon>0)$

例1. $T(n) = 9T(n/3) + n$

解: $a=9$. $b=3$. $n^{\log_b a} = n^2$

$f(n) = n$. $n^2 > n$ // 大 n^1

$\therefore T(n) = \theta(n^2)$ // 情况1 case 1

例2: $T(n) = T(2n/3)+1$

解: $a=1$ $b = 3/2$. $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1 = \theta(1)$

$f(n) = \theta(1)$

$\therefore T(n) = \theta(\lg n)$ // 情况2 case 2

例3. $T(n) = 3T(n/4) + n\lg n$

$\therefore n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$

$f(n) = n\lg n$ // 情况3. case 3

$\because f(n) = \Omega(n^{\log_4 3 + \varepsilon})$

$a f(n/b) = 3(\frac{n}{b})\lg\frac{n}{b} = 3(\frac{n}{4})\lg\frac{n}{4} \le \frac{3}{4}n \lg n = c\, f(n)$

对足够大的 n 成立.

$\therefore T(n) = \theta(n\lg n)$ // case 3.

No.

Date

例4. $T(n) = 2T(n/2) + n\lg n$

$n^{\log_b a} = n < f(n) = n\lg n$

对 $\varepsilon>0$. $\lim_{n\to\infty} \frac{n^{\varepsilon}}{\lg n} = \infty$.

此种情况落于 case 2 与 case 3 之间.

不能使用 master 定理.

Ch5. 概率分析与随机算法

5.1 雇佣问题:

·问题: 中介公司推荐面试者

Interview 中介费 成本小

hire 雇佣 成本大

伪码:

```
Hire Assistant (n) {
    best ← 0 ; //  初值  质量最du
    for i ← 1 to n  do {
        interview i ;
        if ( 候选者 i 比 best 更好 ) {
            best ← i ,
            hire i ;
        }
    }
}.
```

·成本分析

C_i 面试成本

C_h 雇佣成本

设有m个人被雇佣 总成本

$O(n \cdot C_i + m \cdot C_h)$ // n: 固定; m: 不固定

最坏case: 质量按递增面试: m=n.

总成本: $O(n \cdot C_i + n \cdot C_h)$

最好case m=1.

期望

概率分析: 前提 须知道输入的分布情况.

例 雇佣问题

① 申请者按随机序参加面试

② 任意两人的质量可比.

rank(i): 申请者i的质量等级

表: {rank(1) ⋯ rank(n)} 是 {1, 2, ⋯ n} 的排列.

质量表随机均匀分布.

n! 排列用的一种出现的概率相等

随机算法:

特点: 算法行为不仅取决于输入, 也依赖于随机数发生器产生的值.

Random (a, b) . 产生a, b之间的某个值.

雇佣问题的概率分析

设随机变量 X, 表示雇佣新人的数目

$E[X] = \sum_{x=1}^{n} x \Pr\{X=x\}$ // 雇佣人数平均数

结论 $\ln n + O(1)$ (自然对数 + 常数)

雇佣总成本: $O(C_h \ln n)$

5.2 随机算法

强迫输入分布是随机分布

与概率分析的区别

① 雇佣方法是确定的；期望成本与特殊实例不同

设质量表为：$A_1 = \{1, 2, \cdots, 9, 10\}$ // 录取10人

$A_2 = \{10, 9, \cdots, 2, 1\}$ // 录取"10"一人

$A_3 = \{5, 2, 1, 8, 4, 7, 10, 9, 3, 6\}$ // 录用"5" "8" "10" 三人

② 对给定的输入实例，被雇佣者人数均是确的

同一实例运算多次，但结果相同

③ 随机算法：同一实例运行多次结果不是唯一的

没有哪一个特殊实例会导致最坏情况发生

```
Randomized HireAssistant (a) {
    随机枚举候选者表：
    best ← 0
    for i ← 1 to n do {
        Interview i;
        if (i 优于 best 质量高) {
            best ← i;
            hire i;  }  }  }
```

期望雇佣成本：$O(c_h \ln n)$

· 随机枚举数组

假定A是值为 1~n 的数组

① 方法一

为每个A[i]指定优先级P[i]

根据优先级对A排序

```
PermuteBySorting (A) {
    n ← length [A];
    for i ← 1 to n do
        P[i] ← Random (1, n³);
    用P作为关键字对A排序;
    return A;
}
```

时间：$O(n \lg n)$

② 方法二

就地枚举

```
RandomizeInPlace (A) {
    n ← length [A];
    for i ← 1 to n do
        A[i] ↔ A[Random(i, n)];
```

5.3 On-line 的雇佣问题

找质量最好者录用

· 原则 (On-line):

面试每个申请者后, 立即决定是否录取. 只知道局部信息.

最小化面试人数, 最大化录取者质量.

· 方法

Score(i)　第i个申请者得分　分数唯一

当已面试j个申请者后, 已知其中最高分, 但不知后n-j个申请者是否有更高分.

· 策略

选正整数k<n, 面试前k个人, 只记分不录用

① 此后出现的第1个最高分 (高于前面)者被录用

② 若后n-k个申请者分数低于先前最高分, 则录用第n个申请者

```
OnlineMaximum (k, n) {
    bestscore ← -∞;
    for i ← 1 to n do        // 只记分 不录用
        if ( score(i) > bestscore )
            bestscore ← score(i);
    for i ← k+1 to n do      // 录用
        if ( Score(i) > bestscore )
            return i;        // 录用i
    return n;                // 录用n
}
```

· 分析:

对不同k值, 录取最优者的概率不相同

设k固定

∨ $M(j) = \max\{score(i)\}$　// 前j个人的最高分　$(1 \le i \le j)$

∨ 事件S: 成功的选到最优者

∨ 事件S_i: 成功的选到最优者且是第i个面试者

$$Pr\{S\} = \sum_{i=1}^{n} Pr\{S_i\}$$　// S_i 互不相交

∵ $Pr\{S_i\} = 0$,　if $1 \le i \le k$

∴ $Pr\{S\} = \sum_{i=k+1}^{n} Pr\{S_i\}$

如何求 $Pr\{S_i\}$

为保证算法成功选到第i个面试者是最优者, 须:

① 最优者确实位置在i上 —— 事件B_i

② 位置k+1至i-1上的申请者未选中 —— 事件O_i

事件B_i: 位置i的分值是否最高
事件O_i: 前i-1的相对次序　} 互相独立

有:

$Pr\{S_i\} = Pr\{B_i \cap O_i\}$　// 独立

$= Pr\{B_i\} \cdot Pr\{O_i\}$

$Pr\{B_i\} = 1/n$ // 最优者出现在任何位置上的可能性相等

$Pr\{O_i\} = \frac{k}{i-1}$ // 位置1～i-1之间最高分也等可能出现在i-1

// 个位置上，但只有最高分出现在前k个位置上

// O_i才发生

$$Pr\{S\} = \sum_{i=k+1}^{n} Pr\{S_i\} = \sum_{i=k+1}^{n} \frac{k}{(i-1)\cdot n}$$

$$= \frac{k}{n}\sum_{i=k+1}^{n}\frac{1}{i-1} = \frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}$$

积分近似：

$$\int_k^n \frac{dx}{x} \leq \sum_{i=k}^{n-1}\frac{1}{i} \leq \int_{k-1}^{n-1}\frac{dx}{x}$$

$$\frac{k}{n}(\ln n - \ln k) \leq Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

选择k使上面的概率下界最大

$(\frac{k}{n}(\ln n - \ln k))' = \frac{1}{n}(\ln n - \ln k - 1)$ //对k求导

令其为0，有

$\ln k = \ln n - 1 = \ln(n/e)$ $k = n/e$

∵上述下界的二阶导数<0，故当$k=\frac{n}{e}$时，下界取最大值

$$\frac{k}{n}(\ln n - \ln k) = \frac{n}{e \cdot n}(\ln n - \ln(n/e)) = \frac{1}{e} \doteq 0.368$$

当$k=\frac{n}{e}$时，最优者被录用概率最大≥ 36.8%

Ⅱ. 排序和顺序统计

· 排序 $\begin{cases} \text{基于比较：} \Omega(n\lg n) \\ \text{非比较：} O(n) \end{cases}$

统计：在n个元素集中找到第i个最小元

$\begin{cases} \text{有序} & O(\lg n) \\ \text{无序} & O(n) \end{cases}$

Ch 6 堆排序（二叉堆）

· 逻辑结构：完全二叉树

· 存储结构：向量

堆性质：

6.5 优先队列

FIFO 队列 —— 先来先服务

优先队列 —— 按重要性提供服务

key —— 表示优先级 key越大 优先级越高

① 插入

② 返回最大者（优先级最高者）

③ 删除最大者

④ 增值操作 优先级变大

1. 插入：

堆值size+1，就将新结点插入 A[size+1]

向上调整，直到满足堆性质或到达根

```
MaxHeapInsert (A, key) {
    size[A]++;            // 堆长加1
    i ← size[A];          // i是新元素的位置
    while i>1 and A[parent(i)] < key do  // i非根且违反堆性质
        A[i] ← A[parent(i)];   // 违反堆序
        i ← parent(i);    // 交换结果上溯至双亲
    }
    A[i] ← key  // 插入
}
```

2. 取最大 key

返回堆顶 A[1].

3. 删除（出队）

① 取堆顶 A[1]

② 将 A[size]（最后一个元素）送出 A[1]

③ size - 1 ⇒ size

④ 将 A[1..size] 调整为堆

```
HeapExtractMax (A) {
    if size[A] < 1 then
        error ("underflow");
    max ← A[1];           // ①
    A[1] ← A[size[A]];    // ②
    size[A]--;            // ③
    heapify (A, 1);       // 将A[1]调整
    return max;
}
```

• 时间 $O(\lg n)$

4. 增值（提高某节点优先级）　• 时间 $O(\lg n)$

```
HeapIncreasekey (A, i, key) {        // 将A[i]值提高到key
    if (key ≤ A[i])
        error ("…");      // 未增值退出
    A[i] ← key;           // 增值
    while (i>1) and (A[parent(i)] < A[i]) do {
                          // i 非根且违反堆序
        A[i] ↔ A[parent(i)];
        i ← parent(i);    // 上溯到双亲
    }
```

注意：插入可由增值操作来完成.

① size +1 → size

② $A[size[A]] \leftarrow -\infty$,　　即小于堆中任何元素.

③ 调用 HeapIncreasekey (A, size[A], key)

Ch. 7　QuickSort　(快速排序)

最坏情况 $O(n^2)$

期望情况 $O(n\lg n)$

基于比较排序时间下界：$\lg n!$

$\approx n\lg n - 1.44n + O(\lg n)$

快速排序平均时间 $1.39\, n\lg n$

· 随机化

强迫输入实例随机分布

```
Randomized Partition (A, p, r)  {// [A[p, r]
    i ← Random (p, r);   // 在 [p..r] 产生随机整数 i
    A[r] ← A[i];         // A[i] 为划分元，与 A[r] 调换
    return Partition (A, p, r);
}
```

Ch 8.　线性时间内的排序.

8.2 计数排序

给定 n个元素均为 1~k 范围内整数　// k 不一定等于 n. 元素值不一定值一

当 $k = O(n)$，则排序时间为 $O(n)$

简单情况：n 个互不相等的整数取值

范围为 1~n. 则：

```
for (i=1, i≤n; i++)
    B[A[i]] ← A[i];        O(n)
```

基本思想：step1. A 中值为 i 的元素个数记录在 C[i] 中.

$A[1..n] \xrightarrow{C[1..k]} B[1..n]$　　即 C[i] 的值是 A 中等于 i 的元素个数.

step2. 将 C[i] 值改为 A 中小于等于 i 的元素个数

则 C[i] 即为值为 i 的终点位置 (输出)

step3. 将 A[j] 依据 C[A[j]] 的值放入正确位置 B[C[A[j]]] 上,

修改 C[A[j]]--,

```
CountingSort (A, B, k) {
    for i ← 1 to k do  C[i] ← 0;       // 初始化
    for j ← 1 to length[A] do
        C[A[j]] ++;                    // step 1
    for i ← 2 to k do
        C[i] ← C[i] + C[i-1];          // step 2
    for j ← length[A] downto 1 do {    // step 3
```

B[C[A[j]]] ← A[j];

C[A[j]] --; // 为下一值相同元素准备正确位置

}

}

· 时间 $T(n,k) = O(n+k)$;

$= O(n)$ if $k = O(n)$

· 特点

· 稳定的

· k值不能太大

8.3 基数排序

d 位数字或字符

for i ← 1 to d do

使用稳定排序对A第i位进行排序 // 线性时间, 计数

· 时间 $\Theta(d(n+k))$ // n: 元素个数, k 基, d: 关键字位数

当 $k = C(n)$, d = 常数 } $T(n) = O(n)$

· d 与 n 无关, 即 d 为常数? 不一定!

设 n 个数的取值范围是 $C \sim n^c$ (c 为 >1 的常整数)

例: 选基为10, 十进制整数, n^c 需要位数:

$$d = \lfloor \log_{10} n^c \rfloor + 1 \approx c \log_{10} n$$ // d与n相关

当 k=10 // k与n无关

· 时间 $T(n) = \Theta(d(n+k))$

$= \Theta(n \lg n)$

· 改进后为线性:

将 n 个数视作基为n, 则 n^c 表示的位数

$d \approx \log_n n^c = C$ // d与n无关的常数

$k = n$

$\therefore T(n) = \Theta(d(n+k))$

$= \Theta(cn)$

$= \Theta(n)$

· 可操作性问题

选基 $k = 2^{\lfloor \lg n \rfloor}$

8.4 桶排序

假定: 输入是[0,1)区间上均匀分布的实数

· 基本思想:

将[0,1)划分为 n 个大小相等的子区间(桶)

每个桶的大小为$\frac{1}{n}$: $[0,\frac{1}{n})$, $[\frac{1}{n},\frac{2}{n})$, ..., $[\frac{k}{n},\frac{k+1}{n})$, ..., $[\frac{n-1}{n},1)$

将 n 个输入元素分配到这些桶中

对桶中元素进行排序, 然后依次连接桶

No.
Date

· 输入：$0 \le A[1 \dots n] < 1$

辅助数组 $B[0 \dots n-1]$ 是一指针数组，指向桶(链表)

· 关键问题：

$[0,1) \xrightarrow{映射} [0,n)$

$\because$ 设 $\frac{k}{n} \le A[i] < \frac{k+1}{n}$

$\therefore$ $k \le nA[i] < k+1$

$\therefore$ $\lfloor n \cdot A[i] \rfloor = k$

```
Bucket Sort (A) {
    n ← length[A],
    for i←1 to n do        // 扫描 A          时间: O(n)
        将 A[i] 插入链表 B[⌊nA[i]⌋] 中,
    for i←0 to n-1  do     // 桶号 0 ~ n-1
        用插入排序将 B[i] 排序
    将 B[0], B[1], ..., B[n-1] 依次链接;   // 时间: O(n)
}
```

· 时间分析：

期望时间：$O(n)$

$\because$ n个元素均匀分布

$\therefore$ 每个桶内的期望元素个数为 1

每次插入排序时间 $O(1)$

n 次插入排序时间 $O(n)$

No.
Date

Ch 9　中值和顺序统计

在n个(互不相同)的元素中，选 ith 最小元

若 $i=n$，选最大值

$i=1$，选最小值

$i=\lfloor (n+2)/2 \rfloor$ 和 $\lceil (n+2)/2 \rceil$，取中值

$n = \begin{cases} 奇数 & 1个中值 \\ 偶数 & 2个中值 \end{cases}$

Input：集A，整数 i　　$1 \le i \le n$

Output：元素 $x \in A$，使 x 恰好大于 A 中 i-1 个其余元素

9.1 最小值，最大值(特殊选择)

· 最小/大值　　n-1 次比较

· 同时求最小，大值

独立选　　比较次数：$(n-1)+(n-2) = 2n-3$

成对输入 x, y

① 比较 x 和 y

② 将 $\min(x,y)$ 与当前最小值比较

③ 将 $\max(x,y)$ 与当前最大值比较

每2个元素需3次比较，共 $\lceil n/2 \rceil$ 对

总比较次数：$3\lceil n/2 \rceil - 2$

第1对元素只需1次比较

9.2 期望时间为线性的选择问题(一般选择)

基本思想:(分治法)

利用快排的随机划分.

① 将当前的搜索区间 $A[p..r] \Rightarrow A[p..q-1] \le A[q] < A[q+1..r]$

② if(i = size[左区间]+1) 〈左区间长度加1〉// $k=q-p+1$=size[左区间]+1

return $A[q]$.

③ if $i<k$ then 继续到左区间中找 ith 最小元

else // $i>k$ 继续到右区间中找 $(i-k)$th 最小元.

④ 终结条件: 当前查找区间长度为1 时, 直接返回该唯一元素即可.

```
RandomizedSelect(A, p, r, i)        // 在 A[p..r] 中找 i th 最小值
    if p = r then return A[p]       // ④
    q ← RandomizedPartition(A, p, r)    // 调用随机算法
    k = q-p+1;                      // A[q] 的位置
    if i = k then
        return A[q]                 // 返回划分元
    else
        if i < k then
            return RandomizedSelect(A, p, q-1, i)   // 左区间中查找
        else return RandomizedSelect(A, q+1, r, i-k)  // 右区间中查找
```

· 时间分析:

划分后两个子问题只需解其一.

① 最好: $T(n) = T(n/2) + n$ 〈左右子区间大小相近〉

$\theta(n)$ 〈定理4.1 case 1〉

② 最坏: 小区间长度1. 大区间长度 $n-1$

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

③ 期望时间:

k 以等概率在区间 $[1..n]$ 之间出现 $(\frac{1}{n})$

$$T(n) \le \frac{1}{n}\Big[\sum_{k=1}^{n} T(\max(k-1, n-k))\Big] + O(n)$$

(子区间时间) (Partition 时间)

9.3 最坏时间为线性的选择算法

保证好的划分, 使用快排中确定性划分算法 Partition$(A,p,r,\underline{x})$

关键: 选择好的划分元 x.

Select 算法步骤 $(n>1)$

step 1: 将 n 个元素分成5个1组, 共 $\lfloor n/5 \rfloor$ 组, 剩余 $n \bmod 5$ 个为1组. } $\lceil n/5 \rceil$ 组

step 2: 用插入排序对各组进行排序. 取其中值.

(若最后一组有偶数个元素, 则取较小的中值)

step 3: 使用 Select 递归地找 $\lceil n/5 \rceil$ 个中值的中值 x.

step 4: 使用 x 作为划分元, 调用修改后的 Partition 算法

设 x 是第k个最小元. 则左边有k-1个元素, 右边有n-k个元素

step 5: 若 $i=k$. 则返回 x,

否则递归地调用Select 在左边找 ith 最小元. $(i<k)$

或递归地调用 Select 在右边找 ith 最小元 $(i>k)$

- 时间分析:

关键(key): n个元素中至少有多少个元素 $> x$?

∵ 在 step 2 中 所选的中值至少有一半大于 x.

∴ $\lceil n/5 \rceil / 2$ 组中除 2 个组外. 每组中有3个元素大于 x

(x 所在组, 最后一组)

因此, 大于 x 的元素至少有:

$$3\left(\lceil n/5 \rceil/2 - 2\right) \geqslant \frac{3}{10}n - 6$$

∴右区间长度至少有 $\frac{3}{10}n - 6$

左区间长度至多有 $\frac{7}{10}n + 6$.

假设 step 5 是在较长区间上进一步查找

设 最坏时间为 $T(n)$

step 1, 2, 4. 时间为 $O(n)$

step 3. $T(\lceil n/5 \rceil)$

step 5. 至多为 $T(\frac{7}{10}n+6)$

$$\therefore T(n) \leqslant \begin{cases} \Theta(1) \\ T(\lceil n/5 \rceil) + T(\frac{7}{10}n+6) + O(n) & \text{if } n > 140 \end{cases}$$

用代入法: 设 $T(n) \leqslant cn$

$T(n) \leqslant c\lceil n/5 \rceil + c\left(\frac{7}{10}n+6\right) + a \cdot n$ // a 为常数

$\leqslant \frac{c}{5}n + c + \frac{7}{10}cn + 6c + an$

$= \frac{9c}{10}n + 7c + an$

$= cn + \left(-\frac{c}{10}n + 7c + an\right)$

$\leqslant cn$ if $-\frac{c}{10}n + 7c + an \leqslant 0$

只要 $c \geqslant 10a\left(\frac{n}{n-70}\right)$ 即可

要求 $n-70>0$. 即 $n>70$

∵ 当 $n>140$,

∴ $\frac{n}{n-70} \leqslant 2$

∴ 选 $c \geqslant 20n$ 即可

Ⅲ 数据结构

Ch 13. 红黑树 (RB树)

BST(二叉排序树)上运算: $O(h)$

· 平衡二叉树 $O(\lg n)$

完全平衡二叉树 — 满二叉树

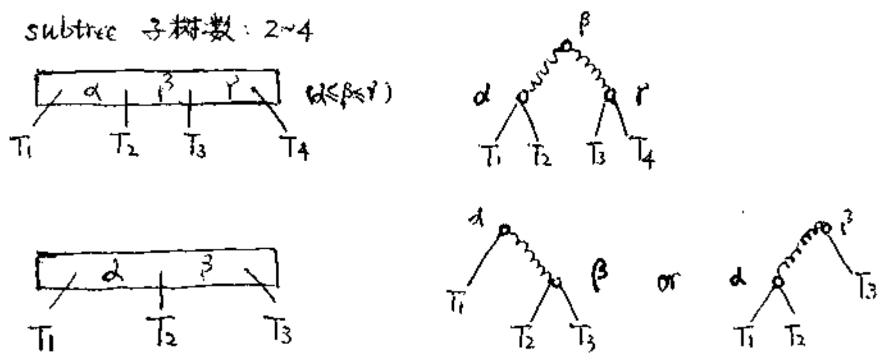
· 平衡的二叉搜索树

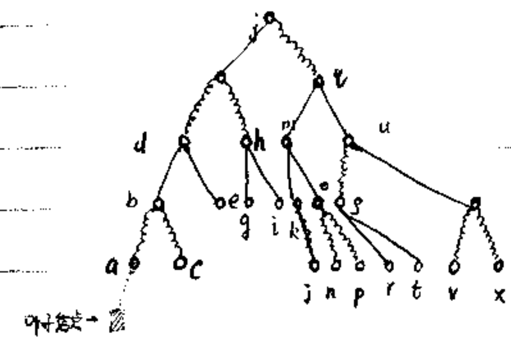
AVL (62年) $1.44\lg n$ (高h至多)

RB树(72年) h至多为 $2\lg(n+1)$

4阶B树 ⟺ RB树

4阶B树 { key 关键字数: 1~3; subtree 子树数: 2~4



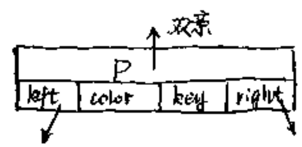


13.1 定义和性质

定义(Def 1) 1: 红黑树是满足下述性质的二叉搜索树 (红黑性质)

1) 每个结点或红或黑
2) 根为黑色
3) 每个叶子(nil)<不含关键字,空指针>: 为黑色
4) 如果结点为红色,则它的两个孩子一定为黑色.
5) 每个结点到其后代叶子所经过的路径含有同样数字的黑色结点

· 结点结构:



内部结点(内点): 含关键字

外部结点(外点): 空指针

· 三种表示: (1) 通常表示 13.1 (a). <P 173>

(2) nil 用哨兵 NIL[T], 只有一个哨兵

(3) 省略空指针

定义2：结点 X 的黑高 bh(x) 是该结点到它的任何后代叶子路径上所经过的黑点数（不包括 X 本身）。

黑高：最少为树高一半，最多为整个树高。

定义3：红黑树的黑高是根的黑高

$bh(root[T])$

Lemma 13.1 一棵 n 个内点（内部结点，不含空指针）的红黑树的高度至多是 $2\lg(n+1)$

证明：方法：设 h 为树高。

① ∵ 红点的下层必为黑。

∴ $bh(root) \geq h/2$. // 黑高至少是整个高度的一半

② 4 阶 B^- 树中每个结点（方框）有一黑色结点，故 B 树高度和 bh(root) 树高一样

RB 内点树 = B 树的关键字数 $n \geq$ 高为 bh(root) 的 B 树结点树（方框数）

$\geq$ 高为 bh(root) 的满二叉树的结点数 $= 2^{bh(root)}-1 \geq 2^{h/2}-1$

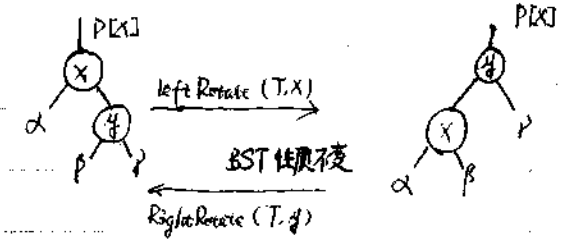
∴ $n \geq 2^{h/2}-1$

$n+1 \geq 2^{h/2}$

$\lg(n+1) \geq h/2$

∴ $h \leq 2\lg(n+1)$

13.2 旋转



从 y 结点为支点旋转

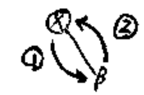
step1: 记录 y

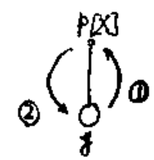
step2: β 作为 X 的右子

step3: y 连在 P[X] 上

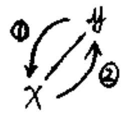
step4: X 连到 y 的左边，作为 y 的左子。

```
LeftRotate (T,X) { // 假定 right[X]（X右子）非空
    y ← right[X];               // step 1.
    right[X] ← left[y];         // step 2. (1)
    p[left[y]] ← X;             // step 2 (2)
    p[y] ← P[X]                 // step 3. (1)
      if P[X] = nil[T] then     // 原X为根
          root[T] ← y;
      else
        if X = left[p(X)] then  // 原X为P[X]左子
           left[p(X)] ← y;
         else
           right[p(X)] ← y;
```





```
    left[y] ← x;      // step 4 ①
    p[x] ← y;         // step 4 ②
}
```



$T(n) = O(1)$ <常数>

13.3 插入

step1: 将z插入到RB树中. z总是作为叶子(不是外部结点)插入.

step2: 将z涂红.

step3: 调整.

```
RBInsert(T, z){
    y ← nil[T];   // y初值
    x ← root[T];  // 从根开始找z的插入位置.
    while x ≠ nil[T] do {   // y是x的双亲
        y ← x;
        if key[z] < key[x] then
            x ← left[x];    // z插入到x的左子树中
        else x ← right[x];  // z插入到x的右子树中
    }     // z作为y的孩子插入.
        p[z] ← y;  // z的双亲为y
    if y = nil[T] then
        root[T] ← z;  // z是插入空树中
    else
        if key[ ] < key[y] then
            left[y] ← z;    // z作为y的左孩子
        else right[y] ← z;  // z作为y的右孩子
    color[z] ← Red;         // z涂红
    RBInsertFixup(T, z);    // 调整
}
```

• 调整分析:

z作为红点, 不违反性质1, 3, 5, 只可能违反性质 2 和 4,

1. 若z作为根插入, 将其涂黑 (恢复性质2)
2. 若z非根, p[z]存在
   ① 若p[z]为黑, 无须调整
   ② 若p[z]为红, 违反性质4, 须调整.

∵ p[z]为红 它不可能为根

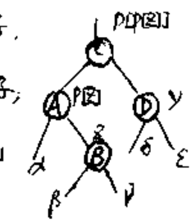
∴ p[p[z]] (z祖父) 一定存在, 且必为黑.

分6种情况调整

case 1-3: p[z]是p[p[z]]的左孩子.

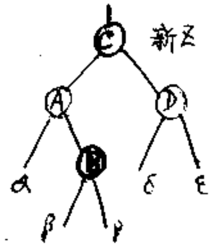
case 4-6: p[z]是p[p[z]]的右孩子;

case 1: z的叔叔(父亲的兄弟) y是红色



(z为p[z]左子树与之类似)

将P[Z]和 y变黑 →
p[p[Z]]变红
令 p[p[Z]] 为新Z

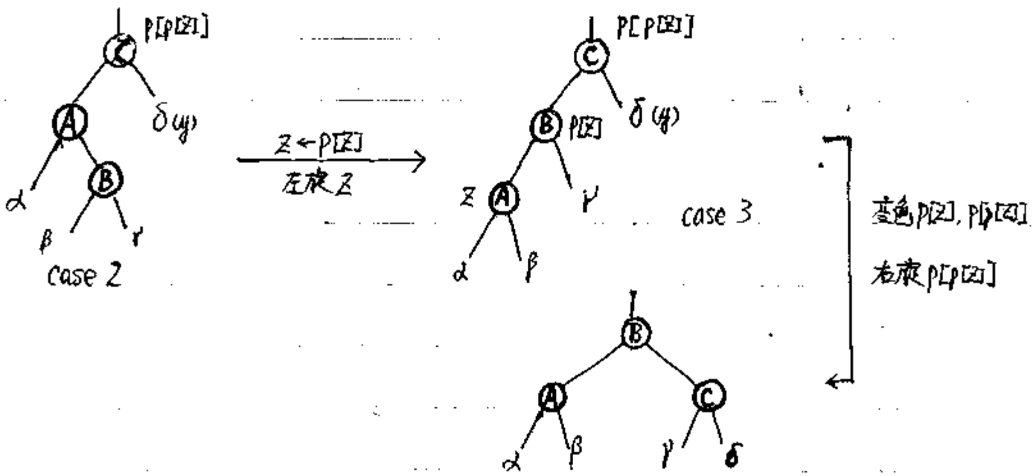


继续保持性质5不变向上调整，最多到根：
- 新Z与Z亲不违反性质4，终止调整
- 新Z为根，涂黑终止

bh + 1

case 2： 当Z的叔叔y是黑色，且Z是p[Z]的右孩子

case 3： 当Z的叔叔y是黑色，且Z是p[Z]的左孩子



终止调整，不用循环

```
RBInsertFixup (T, Z) {    // Z为红
    while ( color[p[Z]] = Red ) do {
        //若Z为根，则p[Z]是nil[T]，为黑，不进循环
        //若p[Z]为黑，无须循环
        if P[Z] = left[p[p[Z]]] then {//case 1-3
            y ← right[p[p[Z]]]; // y为 的叔叔
            if color[y] = Red then { // case1，y为红
                color[p[Z]] ← Black;
                color[ y ] ← Black;
                color[p[p[Z]]] ← Red;
                Z ← p[p[Z]]; // 新Z，红色向上传播，可能循环
            } else { // y为黑， case2 or case 3
                if Z = right[p[Z]] then { // case 2
                    Z ← p[Z];
                    LeftRotate (T, Z);
                }; // 以下为 case 3
                color[p[Z]] ← Black,
                color[p[p[Z]]] ← Red,
                RightRotate (T, p[p[Z]]);
}
```

No.

Date

} else { // case 4-6 与 case 1-3 对称 }

} // endwhile

color [root[T]] ← Black;

}

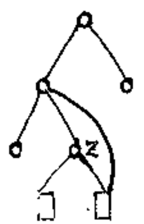
$T(n) = O(\lg n)$

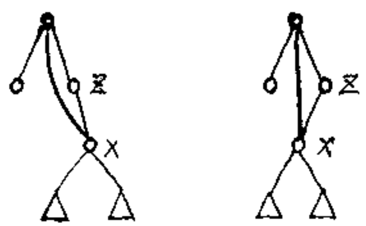
整个插入时间 { 插入 $O(\lg n)$
调整 $O(\lg n)$ }

13.4 删除

在BST上删除，设删Z

case1. 叶子　　　　case2 Z有一个孩子非空

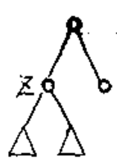




直接删去Z

case 3. Z的两个孩子均非空

删中序遍历后继y.（Z的子树中左子树为空）

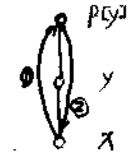


用X取代y.（y的内容拷贝到Z）

No.

Date

```
RBDelete (T, Z) {
    if (left[Z] = nil[T]) or (right[Z] = nil[T]) then  //case1, 2
        y ← Z        // Z至多有一孩子为空
    else             // case 3
        y ← TreeSuccessor(Z);   // y是Z的中序后继
    if left[y] ≠ nil[T] then    // y至多有一非空的孩子
        x ← left[y];            // x指向y的非空孩子(左孩子)
    else
        x ← right[y];           // x指向y的非空右孩子或空孩子
    // 以下用x取代y，与y的双亲进行链接
        p[x] ← p[y];            // ①. 前提是有哨兵存在
        if p[y] = nil[T] then   // y为根
            root[T] ← x;        // x变为根
        else                    // y不为根
            if y = left[p[y]] then  // y为其双亲的左孩子
                left[p[y]] ← x ;    // x为p[y]的左孩子
            else
                right[p[y]] ← x;    // x为p[y]的右孩子
    if y ≠ Z then               // case 3
        y和Z的数据交换
```



No.

Date

if color[y] = Black then

RBDeleteFixup (T, x), // 调整x的颜色

return y;

}

想像将y的黑色涂到x上

若x原为红 则变黑

若x原为黑 则变"双黑"

① 若x为根，直接移去一层黑（树黑高 -1）

② 若x原为红，将y的黑色直接加在x上（x变黑）

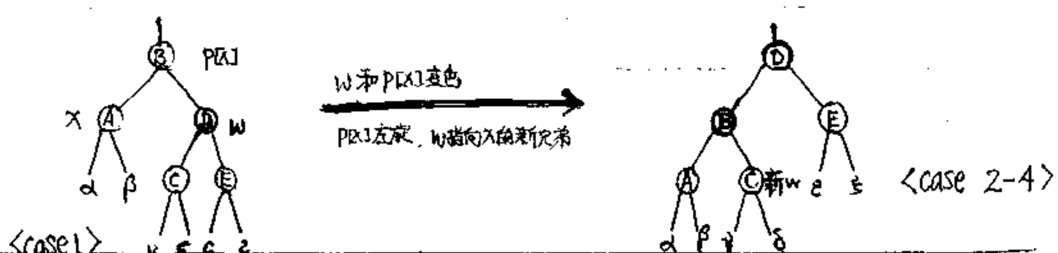
③ 若x非根且原为黑，则通过变换（变颜色、旋转），将x上多余的一层黑向树上方移动，直至x指向某红点或根时终止

共分为8种情况

case 1~4: x 为 p[x] 的左子

case 5~8: x 为 p[x] 的右子

case 1: x 的兄弟 w 是红色

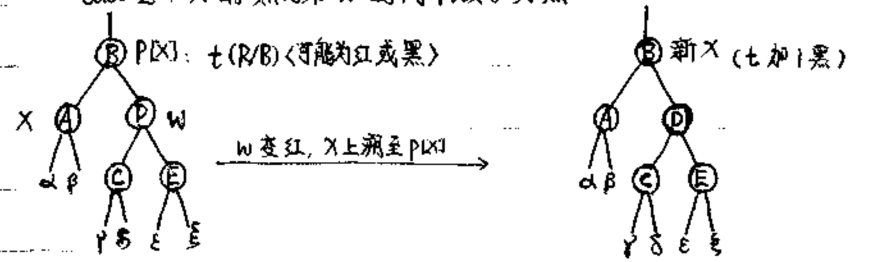


No.

Date

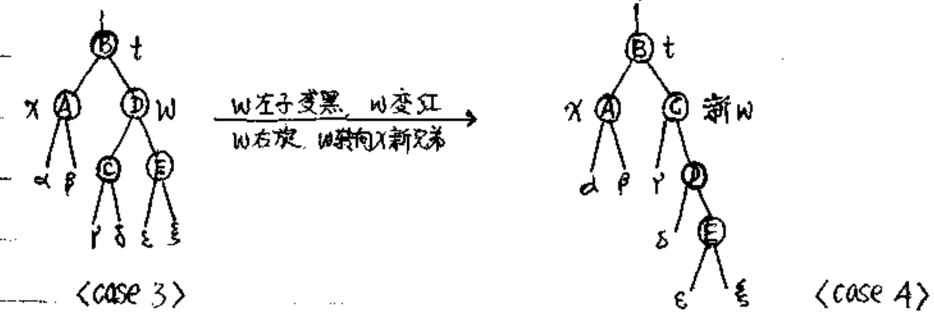
case 2-4: x的兄弟w为黑色

case 2: x的黑兄弟w的两个孩子为黑

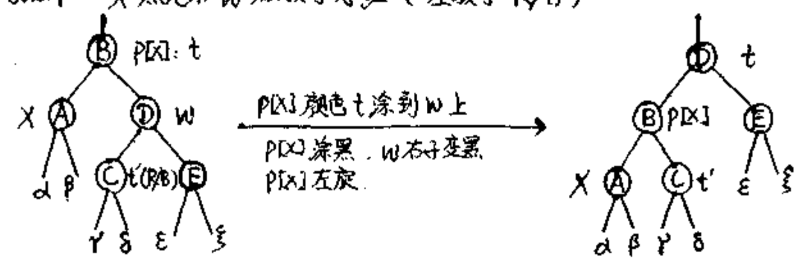


可能循环 { t=Black 继续循环; t=Red 调整结束 }

case 3: x 黑兄弟w的右孩子为黑（左孩子为红）



case4: x黑兄弟w右孩子为红（左孩子 R/B）



x由"双黑"变成"单黑"，调整终止

· 时间 $T(n) = O(\lg n)$

```
RBDeleteFixup(T,X){                //参考前面插入部分算法.
  while (X≠root[T]) and (color[X] = Black)
      do {
           …
         }
      color[X] = Black;
  }
```

· 至多用3个旋转.

Ch 14 数据结构的扩张

目的: 增加新操作, 加速已有的操作

维护: 有效维护新增属性.

14.1 动态顺序统计

通过修改RB树, 使之支持.

1. 选择问题: 给定一个Rank, 求给定集合的相应元素.
2. Rank问题: 求给定元素X在集合中的秩.

顺序统计树(OS树):

OS树 是一RB树, 每个结点上扩充一域: size[X], 它是以X为根的子树中内部结点的总数(包括X自己). 即子树的 size.

设 size[nil[T]] = 0　　叶子的size为0

则 size[X] = size[left[X]] + size[right[X]] + 1

OS树结点结构:

key
size

1. 选择问题　OSSelect(X, i)

是在以X为根的子树中找第i个最小元.

在T中找第i个最小元, 将令X为root[T]即可.

No.
Date

step 1. $r \leftarrow size[left[x]] + 1$ // 在以x为根的子树中 x 的秩
step 2: ① 若 $i=r$，则 返回x
② 若 $i<r$，则 递归地在x的左子树中继续找 i 的最小元.
③ 若 $i>r$，则 递归地在x的右子树中继续找 $(i-r)$ 的最小元.

· 时间 $T(n) = O(\lg n)$

2. 确定给定元素 x 的秩.（在整个树T中）

step 1: 在以x为根的子树中，排在它前面的结点数
$r \leftarrow size[left[x]] + 1$ // r为在以x为根的子树中x的秩
step 2: ① 若x是T的根，则返回 r
② 若x是其双亲的左孩子，
将计算结点x上移至 p[x]，r 也是在以p[x]为根的子树中的秩
③ 若x是其双亲p[x]的右孩子
$r \leftarrow r + size[left[p[x]]] + 1$
计算点上移至 p[x]
r为 x 在以p[x]为根的子树中的秩
重复②③，直至上溯到根，则 r 才是x在整棵树T中的秩.

```
OSRank (T, x) { // x为内部结点，哨兵size = 0
    r ← size[left[x]] + 1,    // step 1
    y ← x ;                   // 计算点 y
```

No.
Date

```
    while y ≠ root[T] do { // 计算点y为根终止
        if y = right[p[y]] then      // step 2 中③
            r ← r + size[left[p[y]]] + 1 ;
        y ← p[y] ;      // 计算点上移
    }
    return r;
}
```

· 时间 $T(n) = O(\lg n)$

3. 维护子树的 size

有效维护？
修改操作：插入，删除.
维护 size 增加成本，但不影响渐近时间.
① 插入
阶段1(插入)：新增成本 $O(\lg n)$
阶段2(调整)：成本 $O(\lg n)$
总成本 $O(\lg n)$
② 删除
阶段1(删除)：

14.2 怎样扩充一个数据结构

1. 选择一个基本数据结构
2. 确定附加信息
3. 确定有效维护附加信息
4. 开发新操作

OS树 —— 在RB树上扩充一个Rank域 《成本太高，失败》

维护Rank的成本： 线性 $O(n)$ （原为 $\lg(n)$）

定理 14.1： （Th14.1）

在RB树上扩充f域，若只与本结点以及其孩子结点有关，则一定有效。

14.3 区间树

· 闭区间：

实数对 $[t_1, t_2]$ 使得 $t_1 \le t_2$

· 开、半开区间

$(t_1, t_2]$ $t_1 < t \le t_2$

$[t_1, t_2)$ $t_1 \le t < t$

· 区间：事件占用的时间

区间 $[t_1, t_2]$ 可表示为对象

它有属性：

$low[i] = t_1$ //起点，低端

$high[i] = t_2$ //终点，高端

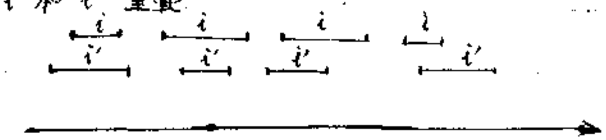
· 两区间重叠 $i \cap i' \ne \emptyset$

$(low[i] \le high[i'])$ and $(low[i'] \le high[i])$

两个区间的低点都不大于另一区间的高点（高端）

· 任两个区间 i 和 i' 均满足区间三分律： 三者其一必成立

① i 和 i' 重叠

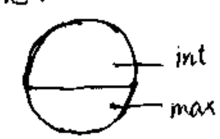


② $high[i] < low[i']$

③ $high[i'] < low[i]$

} 不重叠

· 区间树是RB树的扩充：



step1 基本结构

选RB树. $\forall x \in T$

① x 包含区间 $int[x]$

② x 的key为区间低点（低端）： $key = low[int[x]]$

step 2. 附加 Info

max [X]：以 X 为根的子树中所有结点区间高端的最大值.

step 3：维护附加 Info（有效?）

max [X] = max (high [int [X]], max [left [X]],

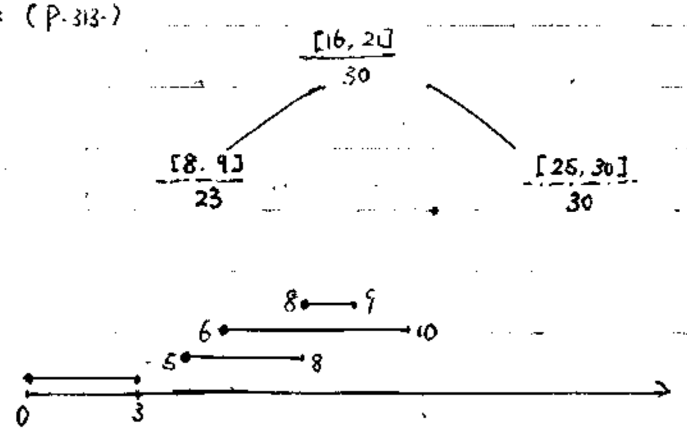
max [right [X]])

满足 Th 4.1（定理4.1 参考前页），则可有效维护 max.

step 4：开发新操作

查找与给定区间 i 重叠的区间 int [X]

例：（P.313）



· 查找与 i 重叠的区间（基本思想）

Interval (Search (T, i):

step 1：从根开始查找 X ← root [X]

step 2：若 X 非空，且 i 与 int [X] 不重叠

1) 若 X 左子树非空，且左子树中最大端 ≥ low [i]

则令 X ← left [X]. // 即在左子树中继续查找

2) 否则，左子树中必定找不到，故令

X ← right [X] // 即在右子树中继续查找

step 3：返回 X // (X = nil) or (int [X] 与 i 重叠)

· 算法的正确性

走一条路径（从上往下）

搜索路径安全.

Th 14.2

case 1：若算法从 X 搜索到左子树，则左子树包含 1 个与 i 重叠的区间，或者 X 的右子树中没有与 i 重叠的区间

case 2：若从 X 搜索到右子树，则左子树中不会有与 i 重叠的区间

pf（证明）

① case 2：

算法走右支的条件：

left [X] = nil or max [left [X]] < low [i]

· X 的左子树为空，则该子树中不会有与 i 重叠的区间

· X 的左子树非空，但 max [left [X]] < low [i]

∵ max [left [X]] 是 X 的左子树中最大端点

∴ ∀ i′ ∈ X 的左子树，有

high [i′] ≤ max [left [X]] < low [i]

∴ 由三分律 i 和 i' 不重叠（ $high[i'] < low[i]$ ）

② case 1：算法走左分支时

· 若 x 的左子树包含与 i 重叠的区间，则搜索该子树是安全的

<case 1 已证明>

· 若 x 的左子树中无区间与 i 重叠，则 x 的右子树中也无区间与 i 重叠

∵ $max[left[x]] \geq low[i]$

∴ ∃ 区间 $i' \in x$ 的左子树，使得：

$$high[i'] = max[left[x]] \geq low[i]$$

又∵ i 和 i' 不重叠 且 $high[i'] < low[i]$ 不成立

∴ 必有 $high[i] < low[i']$ // 由三分律得到

又∵ 区间树中 key 是所有区间的低点，由 BST 性质

∀ 区间 $i'' \in x$ 的右子树 则：

$$high[i] < \underbrace{\underset{\substack{\uparrow \\ key(左子树)}}{low[i']} \leq \underset{\substack{\uparrow \\ key(右子树)}}{low[i'']}}_{(由BST性质得)}$$

$\Rightarrow high[i] < low[i'']$

∴ 由三分律知 i 和 i'' 不重叠。

Ⅳ 高级设计与分析技术

ch 15 动态规划

优化问题求解

四个步骤：

step1：描述最优解的结构特性

2：递归地定义一个最优解的值

3：自底向上计算一个最优解的值

4：从已计算的信息中构造一个最优解

}基础 —— 增加对附加信息的修改

15.1 矩阵链乘

· 设 $<A_1, A_2, \cdots, A_n>$ 是矩阵序列

计算积：$A_1 \cdot A_2 \cdots \cdot A_n$

· 矩阵积的完全括号化：

它是单个矩阵

或是两个完全括号化的矩阵积被包括在一个括号里

· 不同括号方式，产生不同的计算成本

例：A_1 ： 10×100

A_2 ： 100×5

A_3 ： 5×50

$((A_1A_2)A_3)$

A_1A_2: $10\times100\times5=5000$

$(A_1A_2)A_3$: $10\times5\times50=2500$

∴ 共 $5000+2500=7500$ 次

$(A_1(A_2A_3))$

A_2A_3: $100\times5\times50=25000$

$A_1(A_2A_3)$. $10\times100\times50=50000$

∴ 共 $25000+50000=75000$ 次

· 矩阵链乘实质是最优括号化问题

给定 $\langle A_1,\cdots,A_n\rangle$ A_i 的维数 $P_{i-1}\times P_i$ $(1\le i\le n)$

插入括号使其完全括号化. 使得数量乘法次数最少.

· 括号数目

$A_1\sim A_4$ 有 5 种

分裂点: $(A_1 \underset{①}{} (A_2 \underset{②}{} (A_3 \underset{③}{} A_4)))$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$

设 $P(n)$ 表示 n 个矩阵可进完全括号数

假定 k 和 $k+1$ 之间为分裂点

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}p(k)\cdot p(n-k) & n>1\end{cases}$$

$p(n)=C(n-1)$ $\Omega(4^n/n^{3/2})$

穷举法无效!

step1: 最优的括号化结构

将 $A_i\cdot A_{i+1}\cdots A_j$ 积 简记为: $A_{i..j}$ $1\le i\le j\le n$

设 $A_{i..j}$ 的最优括号化的分裂点为 k 和 $k+1$ 之间

分裂点: $i\le k\le j-1$ $(i<j)$

对分裂点 k. 先计算 $A_{i..k}$. 然后计算 $A_{k+1..j}$.

最后计算两子积相乘产生 $A_{i..j}$.

最优括号化的计算成本

计算 $A_{i..k}$ 的成本 + 计算 $A_{k+1..j}$ 的成本 + 两子积相乘的成本.

· 关键:

$A_{i..j}$ 最优括号化 要求

前后缀子链 $A_{i..k}$ 和 $A_{k+1..j}$ 也是最优括号化.

step 2: 一个递归解. (递归定义最优解的值)

用子问题的最优解构造原问题的最优解

确定 $A_{i..j}$ 的最小代价.

设 $m[i,j]$ 是计算 $A_{i..j}$ 的最小乘法次数.

(最优解的值)

① 若 $i=j$，$m[i,j]=0$　$1\le i\le n$

② 若 $i<j$，由 step 1 知

假定最优括号化的分裂点为 k（$i\le k\le j-1$），则：

$$m[i,j] = m[i,k] + m[k+1,j] + P_{i-1}\times P_k \times P_j$$

在不知k取何值时，可在j-i个值（$i\le k\le j-1$）选最优者

$A_{i\cdots j}$ 的最优解的值为：

$$m[i,j]=\begin{cases} 0 & \text{if } i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+P_{i-1}\times P_k\times P_j\} & \text{if } i<j \end{cases} \quad (式1)$$

若要构造最优解，可定义 $S[i,j]$ 记录分裂点 k

step 3：计算最优解值

若简单用递归算法求 $A_{1\cdots n}$，时间为指数

但满足 $1\le i\le j\le n$ 的 i 和 j 共有

$$\underset{i<j}{\binom{n}{2}} + \underset{i=j}{n} = \Theta(n^2) \quad //子问题个数$$

• 算法：

按链长 $j-i+1$ 递增 计算 $m[i,j]$

输入：$P=\langle P_0, P_1, \cdots, P_n\rangle$　A_i 的维数 $P_{i-1}\times P_i$

$m[1..n, 1..n]$ 和 $S[1..n, 1..n]$ 分别记录

最优解的值 和 相应的分裂点

```
MatrixChainOrder(p) {
    n ← length[p]-1;
    for i ← 1 to n do   //链长为1 (矩阵对角线)
        m[i,i] ← 0;
    for l=2 to n do   // l为链长，分别为 2…n
        for i ← 1 to n-l+1 do {
            j ← i+l-1;   //Ai..j长度为l，j-i+1=l
            m[i,j] ← ∞;   // m[i,j]初始化
            for k ← i to j-1 do {   //分裂点
                q ← m[i,k] + m[k+1,j] + Pi-1 × Pk × Pj
                if q < m[i,j] then {
                    m[i,j] ← q;
                    S[i,j] ← k;
                }
            }
        }
    return m & s
}
```

$l=2$ 时，$m[1,2]$，$m[2,3]$，…，$m[n-1,n]$

$l=3$ 时，$m[1,3]$，$m[2,4]$，…，$m[n-2,n]$

No.
Date

· 时间 $T(n) = O(n^3)$

step 4: 构造一个最优解

$S[i,j] = k$ 表示 $A_{i..j} \Rightarrow A_{i..k} \times A_{k+1..j}$

```
MatrixChainMultiply (A, S, i, j) {
    // 求 A1..n  参数 i=1, j=n
    if j>i then {
        X ← MartrixChainMultiply (A, S, i, S[i,j]);
        Y ← MartrixChainMultiply (A, S, S[i,j]+1, j);
        return MartrixChainMultiply (X, Y);  // 通常乘法
    } else return Ai  // i=j
}
```

15.2 动态规划要素

最优子结构 重叠子问题

1. 最优子结构:

$A_{i..j}$ 最优括号化蕴含两个子问题 $A_{i..k}$ 和 $A_{k+1..j}$ 的解也必须最优

· 如何发现最优子结构

① 问题的解必须进行某种选择

② 假定导致最优解的选择已给定

③ 决定产生哪些子问题

④ 最优解内部子问题的解亦为最优

(反证法证明)

· 如何描述子问题空间

$A_{1..n} \Rightarrow A_{1..k} \cdot A_{k+1..n}$

$A_{i..j} \quad 1 \le i \le j \le n$

· 动态规划算法的运行时间

① 子问题总数	最多	$\theta(n^2)$	$\Big\}\ \theta(n^3)$
② 对每个子问题涉及多少种选择		$n-1$	

2. 细节

· 最短路径 (有向无权图)

包含最优子树结构

p: 从 u 到 v 的最短路径, 设 w 是 p 的中间点

$u \overset{p}{\leadsto} v \Rightarrow u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$

显然 p_1 和 p_2 亦须最短 (反证法证明)

· 最长路径 (有向无权图)

$u \overset{p}{\leadsto} v \Rightarrow u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$

p 最长 ⇒ p_1 p_2 亦最长 ?

$q \to r \to t$ 是从 q 到 t 的最长路径

但 q 到 r 的最长路径: $q \to s \to t \to r$

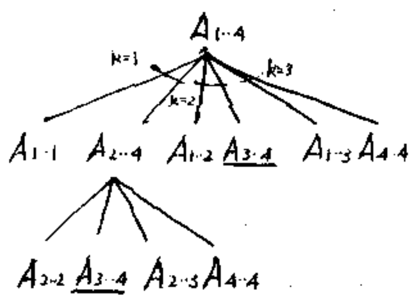
r 到 t 的最长路径: $r \to q \to s \to t$.

子问题的解相互独立，彼此无公享资源.

3 重叠子问题

例: $m[i,j] = \min\limits_{i \le k \le j-1} \{ m[i,k] + m[k+1,j] + P_{i-1}P_kP_j \}$

用自然递归求解时的递归树. ($A_{1..4}$)



$A_{2..4}$: 重叠子问题

4. 重构最优解

附加表保存中间选择结果

5. Memoization

记忆型递归算法 — 动态规划变种

将子问题的解记录在表中

填表和查表

特点:

① 每个问题(子问题)对应一个表项

② 每表目初始为一特殊值表示尚未填入.

③ 递归算法执行过程中, 第一次遇到某子问题时, 计算其解并填入表中,
以后再遇到该子问题时, 将表中值简单返回 (截断递归).

```
MemoizedMatrixChain (p) {            // p: 维数
    n ← length [p]
    for i ← 1 to n do
      for j ← i to n do
        m [i,j] ← ∞ ;                // 初始化. (上面矩阵三角)
    return LookupChain (p, 1, n);
}

LookupChain (p, i, j) {        // 求 Ai..j 的最优解值
   if m[i,j] < ∞ then          // 已经计算过
      return m[i,j];           // 截断递归
   // 第1次遇到 Ai..j, 计算之
   if i=j then m[i,j]=0
   else                        // i<j
```

```
for   k ← i to j-1 do {     // k为分裂点
    q ← LookupChain(p, i, k) + LookupChain(p, k+1, j)
          + p_{i-1} p_k p_j ;
  if q < m[i,j]  then
      m[i,j] ← q ;
 // end for
 return  m[i,j];
}
```

· 时间:

初始化: $\theta(n^2)$

$\theta(n^2)$表目要计算. 每个表目的计算时间为线性 $O(n)$

∴ 总时间: $\theta(n^3)$.

15.3 最长公共子序列 (LCS)

1. 子序列

B是A的子序列. 若B是A中删去某些元素(可不删) 即B不一定是A的连续元素构成的子序列.

·定义:

给定序列 $X = \langle x_1, x_2, \cdots, x_m \rangle$ 序列 $Z = \langle z_1, \cdots z_k \rangle$ 是X的一个子序列, 须满足:

若X的索引中存在一个严格递增的序列 $i_1 < i_2 < \cdots < i_k$ 使得对所有的 j $(1 \le j \le k)$ 均有 $x_{i_j} = z_j$ 成立

2. 两个序列的公共子序列 (CS)

若 Z是X的子列 } ⇒ Z是X和Y的公共子序列(CS)
若 Z是Y的子列 }

3. 最长的公共子序列

X和Y的CS中长度最大者

4. 如何求两给定序列的LCS

step 1. 刻化LCS的结构特征

· 穷举. (X)

定义X的ith的前缀 $X_i = \langle x_1, x_2, \cdots, x_i \rangle$

$1 \le i \le m$, 定义 $X_0 = \phi$

定理 Th 15.1 (一个LCS最优子结构)

设 $X = \langle x_1, x_2, \cdots x_m \rangle$ 和 $Y = \langle y_1, y_2, \cdots, y_n \rangle$ 是序列,

$Z = \langle z_1, z_2, \cdots, z_k \rangle$ 是X和Y的任意一个LCS

(1) 若 $x_m = y_n \Rightarrow z_k = x_m = y_n$ 且 Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS

(2) 若 $x_m \ne y_n$ 且 $z_k \ne x_m \Rightarrow$ Z是X_{m-1}和Y的一个LCS.

(3) 若 $x_m \ne y_n$ 且 $z_k \ne y_n \Rightarrow$ Z是Y_{n-1}和X的一个LCS.

step 2. 子问题的递归解

Th 15.1 ⇒ 寻找X和Y的一个LCS分解为:

① if $X_m = y_n$ then 需解1个子问题：找 X_{m-1} 和 Y_{n-1} 的一个LCS

② if $X_m \neq y_n$ then 需解2个子问题：$\left\{\begin{array}{l}\text{找 } X_{m-1} \text{ 和 } Y \text{ 的一个LCS} \\ \text{找 } X \text{ 和 } Y_{n-1} \text{ 的一个LCS}\end{array}\right\}$ 二选一

用 C 记录最优解的值

$C[i,j]$ 定义为 X_i 和 Y_j 的一个LCS的长度 $0 \le i \le m$, $0 \le j \le n$

$$C[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \quad X_i \text{和} Y_j \text{有一个为空} \\ C[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } X_i = Y_j \quad // \text{case 1} \\ \max\{C[i,j-1], C[i-1,j]\} & \text{if } i,j>0 \text{ and } X_i \neq Y_j \quad // \text{case 2} \end{cases}$$

step 3、计算最优解

子问题空间规模 $\theta(m \cdot n)$

· 输入 $X = \langle X_1 \ldots X_m \rangle$ $Y = \langle Y_1 \cdots Y_n \rangle$

· 输出：$C[0..m, 0..n]$ —— 最优解的值

$b[1..m, 1..n]$ —— 解矩阵

$$b[i,j] = \begin{cases} \text{"}\nwarrow\text{"} & \text{if } C[i,j] \text{ 由 } C[i-1,j-1] \text{ 确定} \\ \text{"}\uparrow\text{"} & \text{if } C[i,j] \text{ 由 } C[i-1,j] \text{ 确定} \\ \text{"}\leftarrow\text{"} & \text{if } C[i,j] \text{ 由 } C[i,j-1] \text{ 确定} \end{cases}$$

```
LCS_length(X,Y)
    m ← length[X];
    n ← length[Y];
    for i ← 1 to m do c[i,0] ← 0;    // 0th列 (第0列)
    for j ← 1 to n do C[0,j] ← 0;    // 0th 列
    for i ← 1 to m do    // 依次考虑 X1, X2, … Xm
      for j ← 1 to n do
        if Xi = Yj then {              // case 1
            C[i,j] = C[i-1,j-1]+1;
            b[i,j] = "↖";
        } else                          // Xi ≠ Yj
            if C[i-1,j] ≥ C[i,j-1] then {
              C[i,j] ← C[i-1,j];
              b[i,j] = "↑";
        } else {
            C[i,j] ← C[i,j-1];
            b[i,j] = "←";
      }
    return b & C;
}
```

· 时间：$\theta(m \times n)$

step 4 构造一个 LCS

从 $b[m,n]$ 开始沿箭头上溯至 $i=0$ or $j=0$ 为止

当 $b[i,j] = $ "↖" 打印 X_i (或 Y_j)

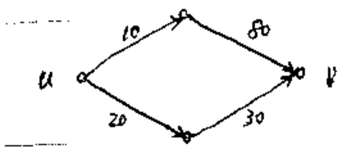
打印算法: Print_LCS (时间: $O(m+n)$)

5. 改进代码:

① b 矩阵可省略: 以时间为代价, 来构造最优解

② c 只要两行: 不能构造最优解

ch 16 贪心法



从 u 到 v: 90 (10+80)

每步都选最好的情况.

16.1 活动选择问题

设 n 个活动 $S = \{a_1, \dots, a_n\}$

均要使用资源 (教室), 独占式使用

· 每活动 a_i 发生时间: $[s_i, f_i)$

满足条件: $0 \le s_i < f_i < \infty$

· 两个活动 a_i, a_j 相容 (不冲突)

$[s_i, f_i)$ 和 $[s_j, f_j)$ 不重叠

判定: $s_i \ge f_j$ or $s_j \ge f_i$

· 活动选择问题:

选最多的互不冲突的活动

选 $A \subseteq S$, 使 A 中活动互不冲突且 $|A|$ 最大.

i	1	2	3	4	5	6	7	8
s_i	1	3	0	5	3	5	6	8
f_i	4	5	6	7	8	9	10	11

问题解: $A_1 = \{a_3, a_9, a_{11}\}$, $A_2 = \{a_1, a_4, a_8, a_{11}\}$

$A_3 = \{a_2, a_4, a_9, a_{11}\}$

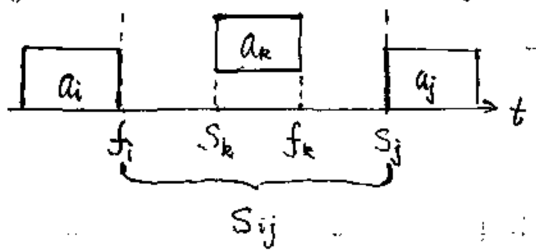
最优解： A_2，A_3.

先按动态规划解法

1. 最优子结构

· 子空间描述：

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$　　// a_k: $[s_k, f_k)$



a_k 与 a_i, a_j 相容

S_{ij} 与 a_i, a_j 相容

· 引入虚拟活动 a_0 和 a_{n+1}

$f_0 = 0$,　$s_{n+1} = \infty$

$S = S_{0,n+1}$

· 假设

$f_0 < f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$　　(16.1 式)

$\because$ $S_{ij} = \phi$　　if $i \ge j$

$\therefore$ 只考虑　$i<j$ 时的 S_{ij}

$0 \le i < j \le n+1$

· 如何分解问题

设子问题 $S_{ij} \ne \phi$　设 $a_k \in S_{ij}$

$f_i \le s_k < f_k < s_j$，若 S_{ij} 的解选择 a_k 之后，导致分解(S_{ij}分解)

① S_{ik}：包括 a_i 完成之后开始，a_k 开始前完成的活动

② S_{kj}：包括 a_k 完成之后开始，a_j 开始之前完成的活动

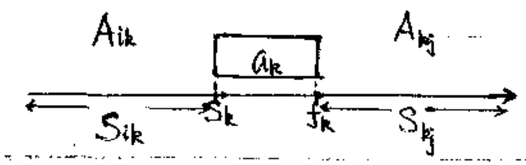
S_{ik} 和 S_{kj} 是 S_{ij} 子集，且与 a_k 相容（不重叠）

S_{ij} 解为 S_{ik} 和 S_{kj} 解的并，再加上 a_k.

· 最优子结构：

设 S_{ij} 的最优解为 A_{ij}，$a_k \in A_{ij}$.

则子问题 S_{ik} 和 S_{kj} 的解为 A_{ik} 和 A_{kj} 也须最优.



$A_{ij} = A_{ik} \cup A_{kj} \cup \{a_k\}$　　(16.2 式)

由子问题最优解构造原问题最优解.

2. 一个递归解

设 $C[i,j]$ 表示 S_{ij} 最优解的值. 即 S_{ij} 中相容活动最大子集的体积（势）

$C[i,j] = |A_{ij}|$

① 当 $S_{ij} = \phi$ 时，$C[i,j] = 0$　　$i \ge j$

② 当 $S_{ij} \ne \phi$ 时，假设 $a_k \in A_{ij}$.

由 16.2 式得：

$C[i,j] = C[i,k] + C[k,j] + 1$

$$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{i<k<j} \{ C[i,k] + C[k,j] + 1 \} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

3. 动态规划到贪心法的转换

定理 16.1：

设S_{ij}是任一非空子集，a_m是S_{ij}中具有最早完成时间的活动

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

则：① 活动 a_m 必定被包含在S_{ij} 的某最优解中

② 子问题 S_{im} 是空集，使S_{mj} 是唯一可能非空子集

证明　先证②：（反证）

设S_{im} 非空，则∃ $a_k \in S_{im}$，使

$$f_i \leq S_k < f_k \leq S_m < f_m$$

∵ a_k 的完成时间先于 a_m，且 $a_k \in S_{ij}$

∴ 与 a_m 是S_{ij} 中最早完成的活动矛盾

再证①：（构造法）

设A_{ij} 是 S_{ij} 某最优解 A_{ij} 中活动已按完成时间增序排列，a_k 是其中 1st 活动

若 $a_k = a_m$，则问题得证，

若 $a_k \neq a_m$，则构造子集

$A_{ij}' = (A_{ij} - \{a_k\}) \cup \{a_m\}$，需证 A_{ij}'是最优解

∵ $a_k \in A_{ij} \subseteq S_{ij}$　∴ $f_m \leq f_k$

又∵ a_k是 A_{ij} 中最早完成的活动　a_m 比 a_k 更早完成

∴ A_{ij}' 中活动互不冲突，它是S_{ij} 的一个解

∵ $|A_{ij}'| = |A_{ij}|$

∴ A_{ij}' 是S_{ij} 的一最优解，它包含 a_m

· 定理价值：

保证贪心法正确性，

简化求解方法

① S_{ij} 分解后只有一个待解子问题

② 只须考虑一种选择：S_{ij} 中最早完成的活动

4. 递归的贪心算法

5. 迭代的贪心算法

```
GreedyActivitySelector ( s,f ) {
  //假定  f1 < f2 < ··· < fn
    n ← length [S];
    A ← {a1};         // A为解集合
    i ← 1 ;           // i是最近加入A的活动的下标
```

```
for  m ← 2  to  n  do       // 找S_{i,n+1}中最早完成的a_m
    if  f_i ≤ S_m then {     // a_m 与 a_i 相容  i<m
        A ← A ∪ {a_m}
        i ← m;
    }  // 否则放弃 a_m.
return  A;
}
```

- 时间：$O(n)$

16.2 贪心策略要点

步骤：

① 确定问题的最优子结构

② 给出递归解

③ 证明在递归的每一步，有一个最优的选择是贪心的选择

④ 证明除了贪心选择导出的子问题外，其余的子问题为空集

⑤ 根据贪心策略写出递归算法

⑥ 将递归算法转换为迭代算法

一般步骤：

① 将优化问题分解为做出一种选择及留下一个待解子问题

② 证明原问题总存在一个最优解会做出贪心选择（保证安全）

③ 验证当做出贪心选择后，它和剩余的一个子问题的最优解组合在一起，构成了原问题的最优解

- 两个要素：

贪心选择性质，最优子结构.

1. 贪心选择性质：

2. 最优子结构：

原问题的最优解 ⇒ 贪心选择 + 1个子问题的最优解.

3. 比较（动态规划，贪心法）

选择？

背包问题：n个物品重 $w_1, \dots, w_n$，背包容量为 $\overline{W}$，

问能否从 n 件物品中选若干件放入背包，使重量之和等于 $\overline{W}$.

1) 0-1 背包，零头背包

n：物品数

i th 的物品价值 v_i，重 w_i 磅（v_i, w_i 为整数）

W：背包载重量

选择物品使包中含有价值最大，且重量和 ≤ $\overline{W}$

1件物品只能装包1次.

0-1背包：拿与不拿（0-1）

零头背包：物品可部分装包.

2) 两个背包问题都有最优子结构

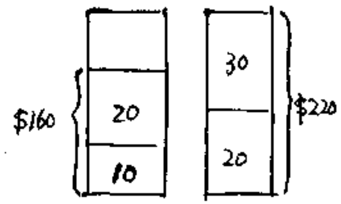
3) 两个背包问题不同解法.

① 0-1 背包（只能用动态规划）

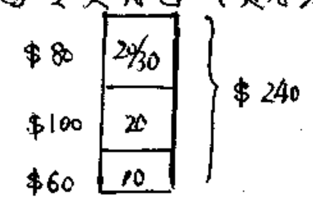
W_i	10	20	30
	\$60	\$100	\$120
V_i/W_i	6	5	4

例：用贪心法：$\overline{W}=50$，$n=3$

·按每磅价值排序（递减）



② 零头背包（贪心法）



Ch 17 平摊分析

在一数据结构上执行一系列操作.

n个操作彼此相关.

·分类法：

① 合计法，② 记帐法 ③ 势能法

·特点：

① 它是一种算法分析的方法，适用一个彼此相关的操作序列.

② 总代价是操作序列长度n的函数

③ 不仅是分析的方法，也是设计算法和数据结构的一种思维方法.

17.1 合计法

对所有的n，具n个操作序列在最坏情况下的总时间为$T(n)$，因此在最坏情况下每个操作平摊代价为$T(n)/n$.

1. 栈操作（不同种类）

·数据结构：栈S，初值为空.

·操作：

Push (S, x) : $O(1)$

Pop (S) : $O(1)$

Multipop (S, k) <从栈顶连续弹出k个元素> : $O(\min(|S|, k))$

<栈大小>

· 栈上操作序列时间分析

① 通常方法

n个操作 Push. Pop. Multipop

一次Multipop 的最坏时间为: $O(n)$

最坏情况Multipop 次数: $O(n)$

∴ 该序列总代价 $O(n^2)$

② 合计法可得紧致界:

∵ 一个对象入栈后至多被弹出一次

∴ 在非空栈上调用 Pop 的次数(包括在 Multipop 中调用)至多是 Push 次数

因此. 当S 初值为空时.

Push 次数至多为 $O(n)$.

Pop 次数也至多为 $O(n)$

又∵ 每个 Push 和 Pop 的实际代价为 $O(1)$

∴ 对任意n. 总时间 $T(n) = O(n)$

每个操作(三种)的平摊分析: $T(n)/n = O(1)$

2. 二进制计数器(同一类操作)

· 数据结构 设 $A[0,\dots,k-1]$ 数组为二进制计数器

初值为 0. A中的 x

$$x=\sum_{i=0}^{k-1} A[i]\cdot 2^i$$

$A[i]$: 二进制位

· 操作: 加 1 (增量), 在模 2^k

$$0 \le x \le 2^k-1$$

```
Increment (A) {
  i←0;
  while ( i< length[A]) and ( A[i]=1 ) do {//从低位开始, 到高位
    // 扫描
      A[i]← 0 ;// 当一位加1, 加1后变0
      i++;        // 进位加到更高位
  };
  if i< length[A] then  // ith 位于 0
      A[i] ← 1 ,  // 进位加到该位上,
```

· 执行过程:

初始为0, n次增量

执行系数	A中值	A[k]…	A[2]	A[1]	A[0]	总成本	本次成本
	0	0 …	0	0	0	0	0
1	1	0 …	0	0	1	1	1
2	2	0 …	0	1	0	3	2
3	3	0 …	0	1	1	4	1
4	4		1	0	0	7	3

No.

Date

· 时间分析:

① 通常分析:

最坏时增量改变的位数为 $O(k)$

n个增量作用初值为0的计数器上，总代价 $O(nk)$

② 合计法:

A[0] 每执行1次翻转1次 共n次翻转

A[1] 每执行2次翻转1次 共 $\lfloor n/2 \rfloor$ 次翻转

A[2] 每执行4次翻转1次 共 $\lfloor n/4 \rfloor$ 次翻转

⋮

A[i] 每执行 2^i 次翻转1次 共 $\lfloor n/2^i \rfloor$ 次翻转

$0 \leq i \leq \lfloor \lg n \rfloor$

∵ n 的二进制表示最多为 $\lfloor \lg n \rfloor + 1$ 位

变位 ($i > \lfloor \lg n \rfloor$) 不翻转

n次增量总的翻转次数为:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} \quad (2n)$$

平摊: $O(n)/n = O(1)$

17.2 记帐法:

· 费用分配 (平摊代价)

No.

Date

数据结构 DS ←—— OP 操作

收费 付费

· 超额收费:

平摊成本 > 实际成本

超额部分作为存款，存在数据结构的某个特定对象上

· 收费不足:

平摊成本 < 实际成本

差额部分由数据结构特定对象上的存款支付

· 各操作平摊代价如何选择？

每个操作的平摊代价是最坏情况下的平均代价，则必须保证对任意长度n的操作序列，总的平摊代价是总的实际代价的一个上界:

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i \quad (\text{对}\forall n\text{成立}, \ \hat{C}_i\text{平摊成本})$$

$$\sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i \geq 0$$

或: 与数据结构相关的总存款在任何时候均非负.

1 栈操作:

	实际代价	平摊代价
Push	1	2
Pop	1	0
MultiPop	min(|S|, k)	0

$\} \ O(1) = \hat{C}_i$

· 正确性分析:

设每个代价单位为1元.

栈 餐馆盘子

Push 1元存款放到入栈的盘子上

Pop 和 Multipop : $\hat{c}_i = 0$ 每个盘子上1元存款正好支付出栈的实际成本

在一时刻, S中盘子数(存款) ≥ 0

2 二进制计数器:

实际 设每位翻转代价 1元

平摊成本: 置位 (0→1) 收费 2 元

复位 (1→0) 0 元

· 正确性

17.3 势能法

存款 : 作为势能保存在整个数据结构上

· 势能, 势差, 势函数

数据结构 D 初态 D_0

第i个操作: OP_i

$$D_{i-1} \xrightarrow{OP_i} D_i$$

势函数: $\Phi(D_i)$: 将每个 D_i 映射为一实数 ($1 \le i \le n$)

势能: 函数值

OP_i 的实际成本 c_i

$$\text{平摊成本 } \hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{势差}} \qquad (17.2\text{ 式})$$

· 势差:

$\Phi(D_i) - \Phi(D_{i-1}) > 0$

$\hat{c}_i$: 超额收费, 势能增加.

$\Phi(D_i) - \Phi(D_{i-1}) < 0$

$\hat{c}_i$: 收费不足, 势能减少.

$\Phi(D_i) - \Phi(D_{i-1}) = 0$

$\hat{c}_i = c_i$

· 势函数选择:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \qquad (17.3\text{ 式})$$

保证正确:

$$\sum_{i=1}^{n} \hat{c}_i \ge \sum_{i=1}^{n} c_i \Rightarrow \Phi(D_n) - \Phi(D_0) \ge 0 \quad (\text{对 } \forall n \text{ 成立})$$

对 $\forall i \in I$:

$\Phi(D_i) \ge \Phi(D_0)$, 通常 $\Phi(D_0) = 0$

∴ 对 $\forall i \in I$:

$\Phi(D_i) \ge 0$

1. 栈操作:

势函数(Φ)：栈中对象数

设初始空栈：$\Phi(D_0)=0$

$\forall i\in I,\ \Phi(D_i)\geq 0$

· 成本分析：

设当前 $|S|=S$，OP_i 是：

① Push：

势差：$\Phi(D_i)-\Phi(D_{i-1})=(S+1)-S=1$

平摊成本：$\hat{C}_i=C_i+\Phi_i-\Phi_{i-1}=1+1=2$

② Pop：

势差：$\Phi_i-\Phi_{i-1}=(S-1)-S=-1$

平摊成本：$\hat{C}_i=1-1=0$

③ Multipop(S, k)：弹出 $k'=\min(s,k)$

势差：$\Phi_i-\Phi_{i-1}=(S-k')-S=-k'$

平摊成本：$\hat{C}_i=C_i-k'=0$

2. 二进制计数器增量操作：

势函数：计数器中'1'的个数

设初始：$\Phi(D_0)=0\quad(A=0)$

$\forall i\in I,\quad \Phi(D_i)\geq 0=\Phi(D_0)$

①实际成本：

设 OP_i 中复位数目为 t_i，置位最多1个.

$C_i\leq 1+t_i$

② 势差：

设 OP_i 之后计数器中'1'的个数为 b_i，即 $\Phi(D_i)=b_i$，则：

$b_i\leq b_{i-1}-t_i+1$

$\Phi_i-\Phi_{i-1}=b_i-b_{i-1}\leq(b_{i-1}-t_i+1)-b_{i-1}=1-t_i$

③ 平摊成本：

$\hat{C}_i=C_i+\Phi_i-\Phi_{i-1}\leq(1+t_i)+(1-t_i)=2$

17.4 动态表

动态存储管理

· 表扩张：表满时插入X，

① 申请更大的表

② 原表考备到新表

③ 释放原表

④ 插入 X

· 表收缩：表中对象小于某数目时，删除X

① 申请更小的表

② 原表考备到新表

③ 释放原表

④ 删除X

∴ 动态表上插入、删除操作序列总成本为 $O(n)$

插入、删除平摊成本 $O(1)$.

· 装填因子

① 非空表:

$$\alpha(T) = \frac{num[T]}{size[T]} \qquad 0 \le \alpha \le 1$$

· num[T]:表中对象数目. size[T]:表的存储空间大小

② 空表:

$T = \phi$ 指 size[T] = 0

定义: $\alpha(\phi) = 1$

$\alpha = 0 \Rightarrow$ num[T]=0, 但 $T \ne \phi$

· 保证浪费空间不致于太大.

$$\frac{\text{未用空间}}{\text{整个空间}} \le \text{某常数}.$$

等价于 α 有一常数下界 $\alpha(T) \ge \beta > 0$

$$\frac{size - num}{size} = 1 - \alpha \le 1 - \beta$$

17.4.1 表扩张

· 启发技术: 扩大时扩大1倍.

· 算法:

```
TableInsert (T, x) {    //开始 num[T] = size[T] = 0
    if size[T] = 0 then {    //初始空表
        table[T] ← 分配1个单元的表.
        size[T] ← 1;
    }
    if num[T] = size[T] then {    //表满, 扩张
        new_table ← 分配 2*size[T]个单元的表.
        将 table[T]中所有表目 copy 到 new_table 中    //表插入.
        释放 table[T];
        table[T] ← new_table;    //地址
        size[T] ← 2 * size[T];
    }
    insert x into table[T];    //基本插入
    num[T] ++;
}
```

· 时间分析:

① 普通方法:

设 n 个插入 初始 $T = \phi$

$$c_i = \begin{cases} 1 & \text{不扩张} \\ (i-1)+1 & \text{扩张} \quad \langle i-1:\text{表插入}\rangle \end{cases}$$

最坏情况下, $c_i = O(n)$, 总代价 $O(n^2)$

No.
Date

② 合计法分析：

· 单个操作的实际代价

$$C_i = \begin{cases} i & \text{if } i-1 = 2^k \quad \text{整数 } k>0 \\ 1 & \text{否则} \end{cases}$$

· 总的实际代价：

$$\sum_{i=1}^{n} C_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n$$

n: 基本插入，$\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$：表插入.

· 平摊代价

$$\hat{C}_i = 3n/n = 3 = O(1)$$

③ 记帐法分析.

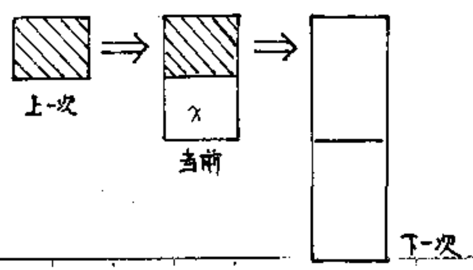
为TableInsert 分配平摊成本3

插入x，收费3元

i) 1元用于x本身插入

ii) 1元存款放于x上，用于支付下次扩张时 copy

iii) 1元存款放于上次扩张到本表的某个元素.



任一时刻总存款>0

No.
Date

④ 势能法

i) 势函数Φ：刚完成扩张时，势能最小(0)

表满时（扩张前一刻）势能最大（num）

$$\Phi(T) = 2 \cdot num[T] - size[T] \qquad (17.5)$$

显然：扩张后一刻，$num = size/2 \Rightarrow \Phi = 0$

扩张前一刻，$num = size \Rightarrow \Phi = size = num$

ii) 正确性：

$\because \alpha \ge \frac{1}{2}$，$num \ge \frac{1}{2} size$

$\therefore \Phi \ge 0 = \Phi_0$.

iii) OP_i 的平摊成本

设OP_i之后表项数，长度及势分别为 num_i，$size_i$ 和Φ_i

显然，$num_0 = size_0 = \Phi_0 = 0$

· 若未引起扩张，则 $size_{i-1} = size_i$

$num_i = num_{i-1} + 1$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$
$$= 1 + (2num_i - size_i) - (2(num_i - 1) - size_i)$$
$$= 3 \qquad // \text{增加 } 3\ 2 \text{ 个单位势}$$

· 若扩张，则 $size_i = 2\, size_{i-1} = 2(num_i - 1)$

$$\hat{C}_i = C_i + \Phi_i - \Phi_{i-1} = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = 3$$

势能减少

17.4.2. 表的扩张和收缩

1. 理想目标

① 表的装填因子 α 有一常数下界 $\beta > 0$

② 表的两种 插入 删除操作的平摊成本有一常数上界

2. 自然策略:

· 表满 ($\alpha = 1$) 时插入, 表扩大一倍

· 表半满 ($\alpha = \frac{1}{2}$) 时删除, 表缩小一倍

满足目标 ①: $\alpha \geq \frac{1}{2}$. 但不满足目标 ②

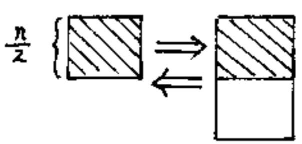
例: 设 $n = 2^k$, 表 $T_0 = \varphi$. 设 n个操作. (I: 插入 D: 删除)

$$\underbrace{I \cdots I}_{2^{k-1}} \ \underbrace{I\,D\,D\,I\,I\,D\,D \cdots}_{2^{k-1}}$$

前一半操作总代价 $O(n)$. 单个操作平摊代价为 $O(1)$

后一半操作, 在第 2^{k-1} 个操作之后表满 $num = size = n/2$

后一半操作 第1个操作引起扩张. 在'DD'操作之后引起收缩. 'II'操作引起扩张, 则进入循环.



每次扩张收缩成本为 $\theta(n)$

收缩 扩张次数, 约 $n/4$.

总代价 $\theta(n^2)$. 每个操作代价为 $O(n)$

原因: 每次基本插入、删除操加势能

1次扩张后, 须做足够删除来积累势能支付下次收缩 copy 成本.

1次收缩后, 须做足够插入来积累势能支付下次扩张 copy 成本.

3. 改进策略:

(1) 设 $\alpha \geq \frac{1}{4}$. 满足目标 ①.

表满 ($\alpha = 1$) 时插入, 扩张1倍.

表 1/4 满 ($\alpha = 1/4$) 时删除, 缩小1倍.

(2) 代码与 TableInsert 类似

(3) 势能分析:

· 势函数:

目标: 最小值(0), 刚完成扩张, 收缩后一刻, $\alpha = \frac{1}{2}$.

最大值:

扩张前一刻: α 由 1/2 增至 1, 势等于表中元素个数 num.

收缩前一刻: α 由 1/2 减至 1/4, 势等于表的 num.

$$\Phi(T) = \begin{cases} 2num[T] - size[T] & 1 \geq \alpha \geq \frac{1}{2} \quad \alpha\text{: 增函数} \\ size[T]/2 - num[T] & \frac{1}{2} \geq \alpha \geq \frac{1}{4} \quad \text{减函数} \end{cases} \quad (17.6)$$

· 正确性:

$\because \Phi_0 = 0$.

① $\frac{1}{2} \leq \alpha \leq 1$. $2\,num \geq size \Rightarrow \Phi \geq 0$

② $\frac{1}{4} \leq \alpha \leq \frac{1}{2}$ $num \leq \frac{1}{2} size \Rightarrow \Phi \geq 0$

(4) 平摊分析:

$num[T] = \alpha(T)\ size[T]$

① OP_i 是插入. $num_i = num_{i-1}+1$

· 若 $\alpha_{i-1} \geq \frac{1}{2}$, 同上节. $\hat{c}_i = 3$

· 若 $\alpha_{i-1} < \frac{1}{2}$, $\alpha_i < \frac{1}{2}$. 不会扩张. 使用17.6 第2段

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0$$

· 若 $\alpha_{i-1} < \frac{1}{2}$, $\alpha_i \geq \frac{1}{2}$.

$$\hat{c}_i = c_i + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) < 3$$

② OP_i 是删除, $num_i = num_{i-1} - 1$

· 若 $\alpha_{i-1} < \frac{1}{2}$. 且 OP_i 不引起收缩 $size_i = size_{i-1}$

$$\hat{c}_i = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$

$= 1 + 1 = 2$ //增加1个单位势能.

· 若 $\alpha_{i-1} < \frac{1}{2}$. 但删除引起收缩 $size_i = \frac{1}{2} size_{i-1}$

实际代价: $c_i = num_i + 1$ (移动 num_i, 删除1项)

$$size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$$

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) = 1$$

· 当 $\alpha_{i-1} \geq \frac{1}{2}$ (Ex 17.4-2)

$\hat{c}_i$ 上界是常数

结论: 每个操作平摊成本 $O(1)$

No.
Date

第五篇 高级数据结构

Ch 19 二项堆

1. 可归并堆

支持5个基本操作

创建, 删除, 存最小关键字, 删最小关键字, 合并

2. 时间比较

二叉堆, 二项堆 Fib 堆

最坏情况 平摊时间

19.1 二项树和二项堆

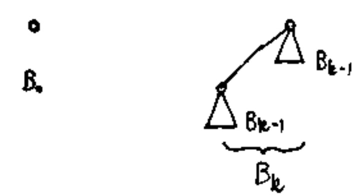
19.1.1 二项树

定义: 仅包含一个结点的有序树是一棵二项树 B_0.

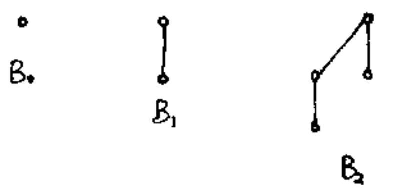
二项树 B_k 由两棵二项树 B_{k-1} 组成, 其中一棵的根作为另一棵树根的最左孩子 ($k \geq 1$)

图19.2:

(a):



(b)

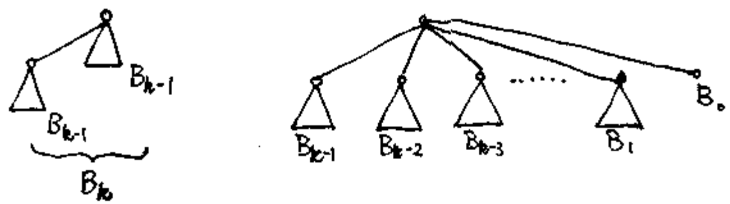


0 层

1 层

2 层（深度为2）

(c)



引理 19.1 （二项树的性质）

二项树 B_k 具有：

1. 有 2^k 个结点
2. 树高为 k　　// $k = \lg n$
3. 深度 i 处恰有 $\binom{k}{i}$ 个结点　　$0 \le i \le k$
4. 根的度最大为 k，若根的孩子从左到右编号为 $k-1, k-2, \cdots, 1, 0$，则子 i 为子树 B_i 的根

证明：对 k 做归纳证明

推论：19.2

在一棵 n 个结点的二项树中，任一结点的最大度数为 $\lg n$.

19.1.2　二项堆

1. 定义：一个二项堆 H 是一个二项树的集合，且满足：

(1) H 中每棵二项树满足最小堆性质.

(2) 对任意非负整数 k，堆 H 中至多有 1 棵二项树根的度数为 k.

性质2：H 中各根的度数各不相同，因此，n 个结点的二项堆 H 中至多有 $\lfloor \lg n \rfloor + 1$ 棵二项树.

证明：n 的二进制表示位数　$\lfloor \lg n \rfloor + 1$

$\langle b_{\lfloor \lg n \rfloor}\, b_{\lfloor \lg n \rfloor - 1} \cdots\cdots b_2\, b_1\, b_0 \rangle$ 使.

$$\underbrace{n}_{\text{结点总数}} = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i \cdot \underbrace{2^i}_{B_i\text{的结点数}}$$

2. 二项堆表示：

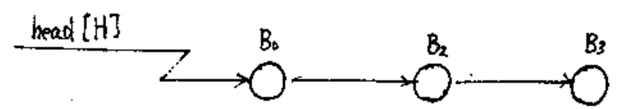
(1) 逻辑表示：

例 $n=13$.　$13_{10} = \langle 1101 \rangle_2$

$B_3\,B_2\;\;B_0$

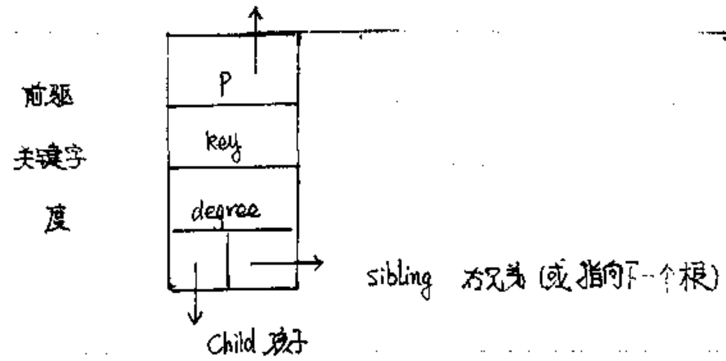
∴ H 中有 B_0, B_2, B_3.

将二项树的根按度数单调增序排成链表（根表）



(2) 存储表示：

左孩子，右兄弟表示，（树的二叉链表



19.2 二项堆上的操作

1. 创建:

分配返回对象H

head[H]←nil

时间: $\theta(1)$

2. 找最小关键字:

算法: Binomial Heap Min (H) {

```
    y ← nil;            //记录当前最小结点
    x ← head[H];  // x是当前扫描结点
    min ← ∞;
    while x ≠ nil do {   //扫描根表.
        if key[x] < min then {
            min ← key[x];
            y ← x;
        }
        x ← sibling[x];
    }
    return y;
}
```

· 时间: 根表长度.

$O(\lg n)$

3. 合并

(1) 基本思想:

· 归并:

head[H] ← Merge (H_1, H_2)

H也是按根度数递增有序

· 合并:

合并H中度数相同的根: 根表度唯一, 有序

Binomial Link (y, z): 将y作为z的最左孩子连到z上.

这里 key[y] ≥ key[z]. 否则 y, z对调

· 合并的实现:

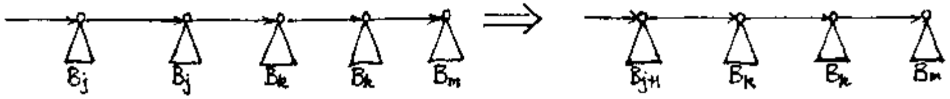
1) 依次扫描H的根表, 将度相同的根合并.

x — 当前检查的根

prev-x — x的前驱

next-x — x的后继

合并过程中, 可能会产生三个相邻的根度相同.



显然 $m > k > j$ $j+1 = k$

合并有4种情况:

case 1:   degree[x] ≠ degree[next-x]   // x及x后继度不相同

  3个指针均后移1个节点.

case 2:   degree[x] = degree[next-x] = degree[sibling[next-x]]

  // 从x起, 连续3个根度相同

  3个指针均后移1个节点.

case 3 & 4:   degree[x] = degree[next-x] ≠ degree[sibling[next-x]]

  // 从x起, 有2个相邻的根度相同

 case 3:   key[x] ≤ key[next-x].

  调用 Link(next-x, x)   // next-x 作为 x 的孩子;

  只需后移一个指针: next-x

 case 4:   key[x] > key[next-x]

  调用 Link(x, next-x)   // x 作为 next-x 的孩子

  后移 x 和 next-x

 当 x是根表的第1个结点时. (prev-x 为空)

  head[H] ⟵ next-x

(2) 算法 Union

 · 时间分析:

归并:

 设 H_1 和 H_2 的结点数分别为 n_1 和 n_2.

  H 的结点数: $n = n_1 + n_2$

  H_1 和 H_2 根表长 $\lfloor \lg n_1 \rfloor + 1$ 和 $\lfloor \lg n_2 \rfloor + 1$

  归并时间: $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \le 2\lfloor \lg n \rfloor + 2 = O(\lg n)$

合并:

  循环次数不超过根表长度 $\le \lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 = O(\lg n)$

∴ 总计: $O(\lg n)$

4. 插入:

 创建新堆 H′, 将 x 插入新堆 H′.

  H ⟵ Union(H, H′)

 · 时间: $O(\lg n)$ // 取决于Union算法

5. 删最小结点:

 ① 基本思想:

  · 查找: 在根表中找到最小结点(key 最小)

  · 逆置: 将 x 的孩子逆置插入一个空堆 H′ 中.

  · 合并: 将 H′ 与删去 x 之后的 H 合并:

    H ⟵ Union(H, H′)

  · 返回 x

 时间: $O(\lg n)$ // 3步都为 $O(\lg n)$.

6. 减值：　将x所指结点的key减小到k

①基本思想：

当 key[x] ≤ k 时，返回或报错，否则：

step1：将 key[x] 减至 k

step2：若违反堆次序，即 key[x] < key[prev[x]]，则

向根方向调整

② 算法

③ 时间：　从x至多调整到根

O(lgn)

7. 删除：　删去任一结点

① 将被删结点x减值到 -∞.

② 删最小结点.

算法时间：　O(lgn)

Ch 20　Fib 堆

· 特点：

Fib 堆结构比较松散

根表中根的度不再唯一

树不再是二项树

根表不一定按度数有序

将所有调整堆的操作延迟

ExtractMin 操作：

20.1　Fib 堆结构

· 定义：　Fib 堆是一个堆有序的树集合.

若不执行 Decrease 和 Delete，则堆中树是无序的二项树.

· 结点结构：

p[x]：　双亲指针

Child[x]：　孩子指针，可指向任一孩子.

right[x]：
left[x]：　} 兄弟指针，双向循环链表（双循环链表）.

① 根表　　② 兄弟链表

degree[x]：　度数（孩子数）

mark[x]：　标记域（true / false）

自x上次成为另一个结点的孩子以来，x是否失去了一个孩子.

当创建新结点及 x 成为另一结点的孩子时. $mark[x]$ 值为 false.

· 堆属性:

$min[H]$: 头指针. 指向根表中 key 最小的结点.

$n[H]$: 堆H中结点的总数.

· 势函数 ϕ:

$\Phi(H) = t(H) + 2m(H)$

$t(H)$: 堆H中树的数目

$m(H)$: 堆H中标记结点的数目.

· 多堆的势能为每个堆势能之和

假定: 一个单位的势可支付常数量 $O(1)$ 的工作.

正确性: 初始堆 $H=\phi$. $\Phi_0=0$.

对任意H. 有 $\Phi(H) \geq 0 = \Phi_0$

· 最大度数.

$D(n)$: n个结点的Fib堆中任一结点度的上界.

若只支付五个基本操作, 则 $D(n) \leq \lfloor \lg(n) \rfloor$

若也支付后两个操作(减支.删除), 则 $D(n) = O(\lg(n))$

20.2 可归并堆操作:

定义: 无序二项树 U_0 包含一个结点. 一棵无序二项树 U_k 包含两棵无序树 U_{k-1}.

其中一棵的根是另一个棵树的孩子.

· U_k 的性质:

① 有 2^k 个结点. ($n=2^k$)

② 树高为 k

③ 深度为 i 处恰有 $\binom{k}{i}$ 个结点 $0 \leq i \leq k$

④ 对于 U_k. 根的度为 k. 它大于其它任一结点的度.

根的孩子按某种次序分别是子树 $U_0, U_1, \cdots, U_{k-1}$ 的根.

由性质①. ④: 若H由含有n个结点的无序二项树构成. $D(n) \leq \lg n$

· 将堆重构尽可能延迟

1. 创建新堆:

令 $n[H] = 0$

$min[H] = nil$

实际成本. 平摊成本: $O(1)$

2. 插入:

将新结点 x 看作 U_0 插入H的根表中.

实际成本: $O(1)$

平摊成本: $\Phi_{k-1} = t(H) + 2m(H)$

$\Phi_k = (t(H)+1) + 2m(H)$

$\hat{C}_k = C_k + \Phi_k - \Phi_{k-1} = O(1) + 1 = O(1)$

3. 找最小结点:

直接返回 $min[H]$ 所指结点. 实际成本: $O(1)$

操作前后势能不变, 平摊成本 $O(1)$

4.合并：

将两根表首尾链接. $C_k = O(1)$

· 时间：

势差：$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$

$t(H) = t(H_1) + t(H_2)$

$m(H) = m(H_1) + m(H_2)$

$\therefore\ \hat{C}_k = C_k$

5.删最小结点：

(1)思想：设 $H \neq \phi$.

step 1：将被删结点z所有孩子作为根插入根表.

step 2：将z从堆H的根表中删去.

若删除前H只有1个结点，则 min[H] ← nil.（step 2.1）

否则，min[H]指向z的后继或z的某个孩子.（step 2.2）

step 3：调整根表：

step3.1：合并度数相同的根

step3.2：确定新的最小结点.

(2)算法：

```
FibHeapExtractMin(H){
    z ← min[H];
    if z ≠ nil  then {    // 若H=φ. 直接返回nil
        for z的每个孩子x do {    // step 1
            将x插入H的根表中;    // 插入左孩子(右孩子)
            p[x] ← nil;
        }
        将z从H的根表中删去    // step 2   z中的值不变.
        if z = right[z]  then    // z是H中的唯一结点
            min[H] ← nil;
        else {
            min[H] ← right[z];    // step 2.2
            consolidate(H);    // step 3
        }
        n[H]--;
    }  //endif;
    return z;
}
```

No.
Date

③ 调整算法

· step 3.1 合并

反复做：

① 在根表中找两个度相同的根 x 和 y，不妨设 key[x] ≤ key[y]（否则 x 与 y 互换）

② 将 y 作为 x 的孩子与 x 链接：

FibHeapLink (H, y, x)： degree[x]++

mark[y] ← false

· 实现 step 3.1：

设置指针数组 A[0.. D(n[H])] 记录当前已检查或处理过的根。

A 的下标为度数，A[i] 的值指向根表中当前已处理的度为 i 的根，即：

若 A[i]=y，则 y 是根表中度为 i 的根，degree[y] = i；

一 A 各分量的初值为 nil

一 w 指向合并过程中当前处理的结点，w 初值为 min[H]，然后依次右链扫描根表，反复做下述 2 个操作，直至 w 指向 min[H] 左边的结点为止

① 若 A[degree[w]] = nil，则 w 是目前为止度数唯一的根，故令：A[degree[w]] ← w.

② 若 A[degree[w]] = y ≠ nil； 即 A 中已有与 w 度相同的根 y，则 y 和 w 合并，产生度为 degree[w] + 1 的根，该根可能引起新的合并，由 A[degree[w]+1] 是否为空来决定；〈此步循环〉

No
Date

step 3.2：重构堆，新堆按根的度有序，确定新的最小结点。

```
Consolidate (H) {
  for i←0 to D(n[H]) do A[i] ← nil ;   //初始化
  for (H 的根表中每个结点 w) do {
     x ← w ;         //当前处理结点
     d ← degree[x] ;  //d 为 x 的度
     while A[d] ≠ nil do {  //A 中已有度为 d 的根，与 x 合并
        y ← A[d] ;   //x 与 y 合并
        if key[x] > key[y] then x ←→ y;
           FibHeadLink (H, y, x);
           A[d] ← nil;
             d ← d+1;    // x 的度已经加1，d 是循环不变量 d = degree[x]
     } //end while     出口时，A 中不再有与 x 度相同的根
     A[d] ← x;   // A[d] 指向当前度为 d 的唯一结点 x
  } //end for
  min[H]←nil;   // 新堆 step 3.2
  for i←0 to D(n[H]) do
     if A[i] ≠ nil then {
        将 A[i] 插到 H 的根表中  //插入 min[H] 左或右
```

No.
Date . .

if (min[H] = nil) or (key[A[i]] < key[min[H]])
then min[H] ← A[i]; // 找最小结点
} end if
}

FibHeadLink

(4) 调整过程 Fig 20.3

(5) H 保持无序二项树集合.

(6) 时间

① 实际成本

step 1: 删最小结点. Z的孩子数 ≤ $D(n)$ <堆最大度数>

step 2: $O(1)$

step 3:

· 初始化: $O(D(n))$

· 合并:

for 循环次数 ≤ $D(n) + t(H) - 1$ <$D(n)$: Z的孩子数. $t(H)$: 原根数>

while 总数不超过根表大小.

总成本: $O(D(n) + t(H))$

· 重构: $O(D(n))$

总成本: $O(D(n) + t(n)) = C_i$

② 平摊成本:

No.
Date . .

$\Phi_{i-1} = t(H) + 2m(H)$

操作后:

至多有: $D(n)+1$ 根, 标记数可能减少

$\therefore \Phi_i \le D(n) + 1 + 2m(H)$

$\Phi_i - \Phi_{i-1} \le D(n) - t(H) + 1$

$\hat{C_i} \le O(D(n) + t(H)) + D(n) - t(H) + 1$ <$C_i + \Phi_i - \Phi_{i-1}$>

$= O(D(n)) + O(t(H)) - t(H)$

$= O(D(n))$

则必须有: $O(t(H)) - t(H) \le 0$

实际成本 势

20.3 减值和删除

使堆破坏无序二项树特性

1. 减值

· 平摊时间为 $O(1)$

· 简单方法

① 若 x 为根, 则减小 $key[x]$ 不调整

② 若 x 非根, 设 $y = p[x]$, 则当 $key[x] < key[y]$ 时, 调整,

将以 x 为根的子树从 y 上删除后插入根表中

时间 $O(1)$, 但会产生问题, 若连续删 y 的孩子

· le.

No

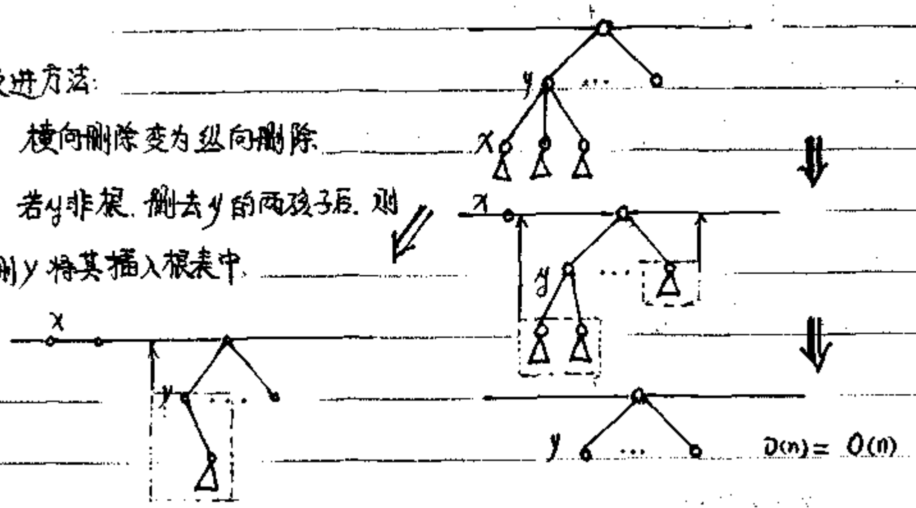
Date

则y所在树与无序二项树差别过大(除非y是根),影响D(n)

改进方法:

横向删除变为纵向删除

若y非根,删去y的两孩子后,则删y将其插入根表中.



若y是双亲z的第2个被删孩子,且z不是根,则z同样删去后插入根表.

这一过程递归直到被删结点的双亲是根,或双亲只失去一个孩子.

如何知道一结点丢掉几个孩子?

mark[y]: 当y非根,失去第1个孩子时,令mark[y] ← true.

· 算法:

```
FibHeapDecreaseKey (H, x, k) {   //将x的关键字减小到k
    if k ≥ key[x] then
        error ("x的key未减小"), //返回
    key[x] ← k;                 //减值
    y ← p[x];
    if (y≠nil) and (key[x] < key[y]) then {  //x非根,违反堆序
        Cut (H, x, y);  //将x从y的孩子中删去,加入根表
        CascadingCut (H, y);  //若x是y的第2个孩子,则可能引起连锁删除
    } //end if
    if key[x] < key[min[H]] then
        min[H] ← x ;
}

Cut (H, x, y) {
    将x从y的孩子链表中删去, degree[y]--;
    将x插入H的根表
    p[x] ← nil;  //x为根
    mark[x] ← false;

CascadingCut (H, y) {  //y刚刚失去了一孩子x
    z ← p[y];
    if z ≠ nil then  //y非根
        if mark[y] = false then  //x是y失掉的第1个孩子
            mark[y] = true;
```

No.
Date . .

else { // x是y失掉的第2个孩子

Cut (H, y, z); // 将y从双亲z的孩子链表中删去

CascadingCut (H, z);

}

}

· 图参考书 P.491 Fig 20.3

· 时间分析:

①实际成本:

Decreasekey 中 除了连锁删外, 其余时间 $O(1)$

设递归调用 CascadingCut 共C次, 每次成本为 $O(1)$

减值总成本: $O(C)$, $C \geq 1$

最坏情况: $C = \lg n$

②势差:

$\Phi_{i-1} = t(H) + 2m(H)$

操作后:

树: 有 $t(H) + C$ 棵

新增: 以x为根树, C次调用新增C I

标记: 连锁C-1次删除的结点的标记由真变假

最后一次调用 CascadingCut 可能增加1标记,

x变为根, 标记可能减少1个.

No.
Date . .

则: $\Phi_i \leq (t(H) + C) + 2(m(H) - (C-1) + 1)$

$\Phi_i - \Phi_{i-1} \leq 4 - C$

③平摊成本:

$\hat{c}_i \leq c_i + 4 - C = O(C) + 4 - C = O(1)$

2. 删除:

时间: $\hat{c}_i = O(1) + O(D(n))$ 减值 + 删最小结点

$= O(D(n))$

20.4 最大度数的界

$D(n) = O(\lg n)$

引入 Fib 级数

练习 Ex 4.1-5
4.2-2
4.3-1
思考题 Prob 3-4.b
3-4.g

(3.25)
4.15

Ex 5.3-6
Ex 8.3-4
8.4-4
Ex 9.2-1

(6.16) Ex 17.3-1
17.3-3

一. 选择题 (4选1)

基本概念, 范围广.

例: 堆的查找效率最低.

二. 简答题:

① 证明: 渐近时间 { 递归 / 求和

② 扩张等操作 表的势能变化 (最大, 最小时刻)

问答

③ 画图.

三. 算法题: (<25分)

注重思路, 程序加注解