第三章 分治法及递归算法的分析方法

3.1 分治法(Divide-and-conquer)

1. 什么是分治法

当求解的问题较复杂或规模较大时，不能立刻得到原问题的解，但这些问题本身具有这样的特点，它可以分解为若干个与原问题性质相类似的子问题，而这些子问题较简单可方便得到它们的解，因此通过合并这些子问题的解就可得到原问题的解。

2. 应用分治法的三个基本步骤

①分解问题**(divide)**：把原问题分解为若干个与原问题性质相类似的子问题

②求解子问题**(conquer)**：不断分解子问题直到可方便求出子问题的解为止

③合并子问题的解**(combine)**：合并子问题的解得到原问题的解

3. 归并排序**(Merge Sort)**

思想：如果能把原待排序的数组分解成若干个待排序的子数组，而这些子数组可以方便地排好序，并且通过合并这些子数组的解将能得到原问题的解，则整个数组将排好序。

应用分治法解题的三个基本步骤为：

①divide：把具有n个元素的数组分解为二个n/2大小的子数组

②conquer：递归地分解子数组，直到子数组只包含一个元素为止

③combine：二二合并已排好序的子数组使之成为一个新的排好序的子数组，重复这样二二合并的过程直到得到原问题的解

由分治法三个步骤可方便得到解此问题的算法：

```
Merge-sort(A,p,r)
    if p<r
    then q← ⌊(p+r)/2⌋
            Merge-sort(A,p,q)
            Merge-sort(A,q+1,r)
            Merge(A,p,q,r)
```

Merge(A,p,q,r)	执行次数
$n_1 \leftarrow q-p+1$	1
$n_2 \leftarrow r-q$	1
for $i \leftarrow 1$ to n_1	$n_1+1 \le n+1$
do $L[i] \leftarrow A[p+i-1]$	$n_1 \le n$
for $j \leftarrow 1$ to n_2	$n_2+1 \le n+1$
do $R[j] \leftarrow A[q+j]$	$n_2 \le n$
$L[n_1+1] \leftarrow \infty$	1
$R[n_2+1] \leftarrow \infty$	1
$i \leftarrow 1$	1
$j \leftarrow 1$	1
for $k \leftarrow p$ to r	$r-p+2 \le n+1$
do if $L[i] \le R[j]$	$r-p+1 \le n$
then $A[k] \leftarrow L[i]$	$\le n_1$
$i \leftarrow i+1$	$\le n_1$
else $A[k] \leftarrow R[j]$	$\le n_2$
$j \leftarrow j+1$	$\le n_2$

4. 归并排序算法分析

首先分析Merge算法的时间：

$\because n_1, n_2 \le n$，$r-p+1 \le n$

$\therefore T(n) = An+B=\theta(n)$

Merge-sort算法为一递归算法，根据递归算法时间函数一般式：

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

其中：$\Theta(1)$表示常数时间

a为分解后的子问题个数

1/b为分解后子问题与原问题相比的规模

D(n)表示分解子问题所花费的时间

C(n)表示合并子问题解所花费的时间

对于Merge-sort算法：

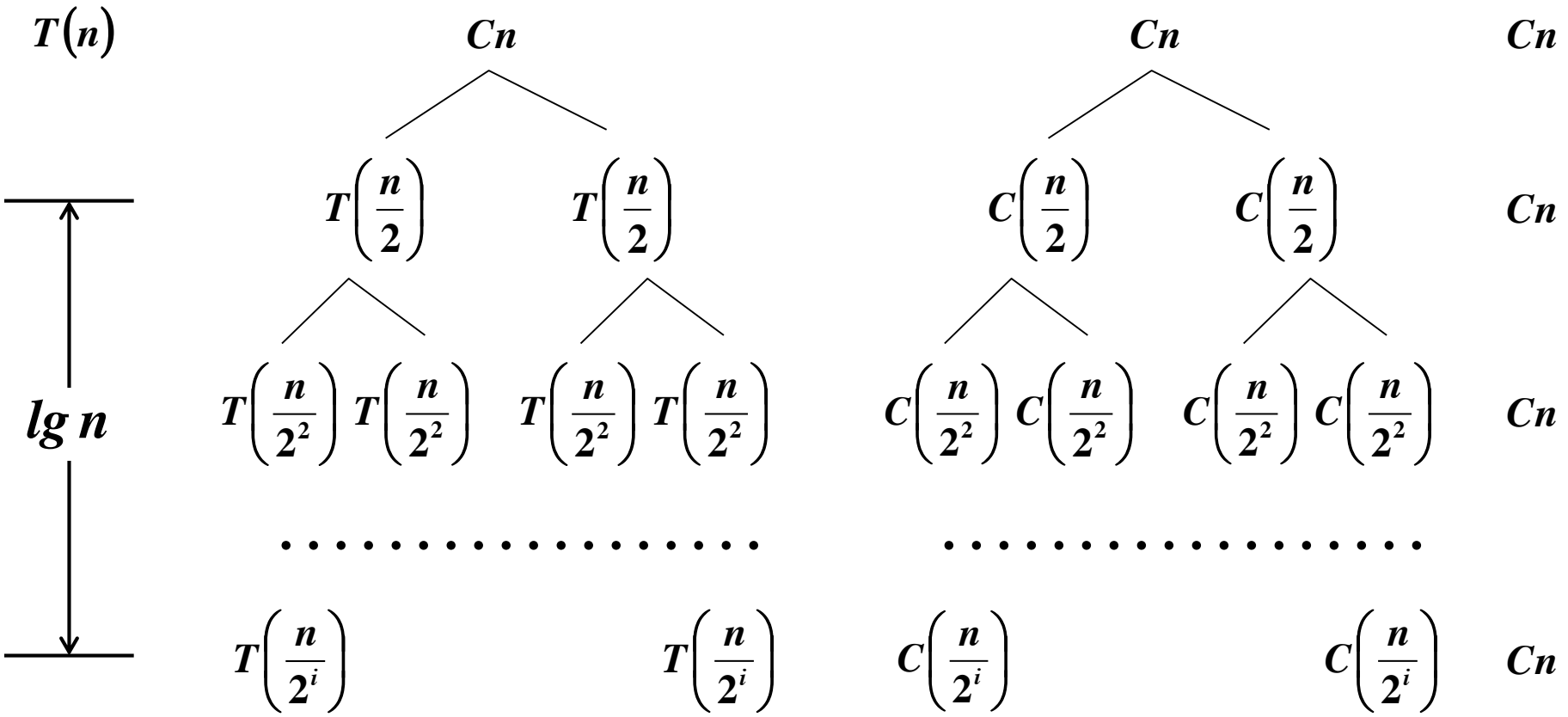
∵ a=2, b=2, D(n)=θ(1), C(n)=θ(n)

$$\therefore T(n)=\begin{cases}\Theta(1) & n=1\\ 2T\left(\frac{n}{2}\right)+\Theta(n) & n>1\end{cases}$$

上式可改为等价的形式：

$$T(n)=\begin{cases}C & n=1\\ 2T\left(\frac{n}{2}\right)+Cn & n>1\end{cases}$$

递归树方法：


T(n)
Cn
Cn
Cn
T(n/2)
T(n/2)
C(n/2)
C(n/2)
Cn
lg n
T(n/2^2) T(n/2^2) T(n/2^2) T(n/2^2)
C(n/2^2) C(n/2^2) C(n/2^2) C(n/2^2)
Cn
T(n/2^i)
T(n/2^i)
C(n/2^i)
C(n/2^i)
Cn

令$n=2^i$，则$i=\lg n$

T(n)=递归树一层的时间×总的层数

$=cn\times(i+1)$

$=cn\times(\lg n+1)$

$=cn\lg n+cn$

数学归纳法证明该结果如下：

proof：归纳基础：当n=1时

$\because T(n)=cn\lg n+cn=c$

$\therefore$ 命题成立

归纳假设：假设$n=2^i$时命题成立

归纳步骤：证$n=2^{i+1}$时命题亦成立

$\because$ $n=2^i$时，递归树共有$i+1$层

$\therefore$ $n=2^{i+1}$时，递归树共有$i+1+1$层

$\therefore$ $T(n)=(i+1+1)\times cn$

$=(\lg 2^{i+1}+1)\times cn$

$=(\lg n+1)\times cn$

$=cn\lg n+cn$

$=\theta(n\lg n)$

递推方法:

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

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

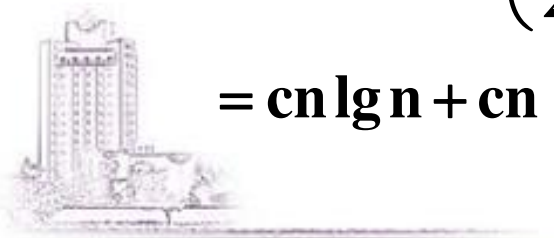
$$= 2cn + 2^2 T\left(\frac{n}{2^2}\right)$$

$$= 3cn + 2^3 T\left(\frac{n}{2^3}\right)$$

$$= \cdots\cdots$$

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

$$= cn\lg n + cn \qquad \text{令} n = 2^i, \text{则} i = \lg n$$



5. 分治法适用条件

①原问题可以分解为若干个与原问题性质相类似的子问题

②问题的规模缩小到一定程度后可方便求出解

③子问题的解可以合并得到原问题的解

④分解出的各个子问题应相互独立，即不包含重叠子问题

如求**Fib**数问题

3.2 递归算法的分析(Recurrence)

三种方法：替换法、递归树方法及Master方法

1. 替换法(Substitution Method)

思想：首先对T(n)的解做一个猜测，然后用归纳法证明所做猜测是否正确。

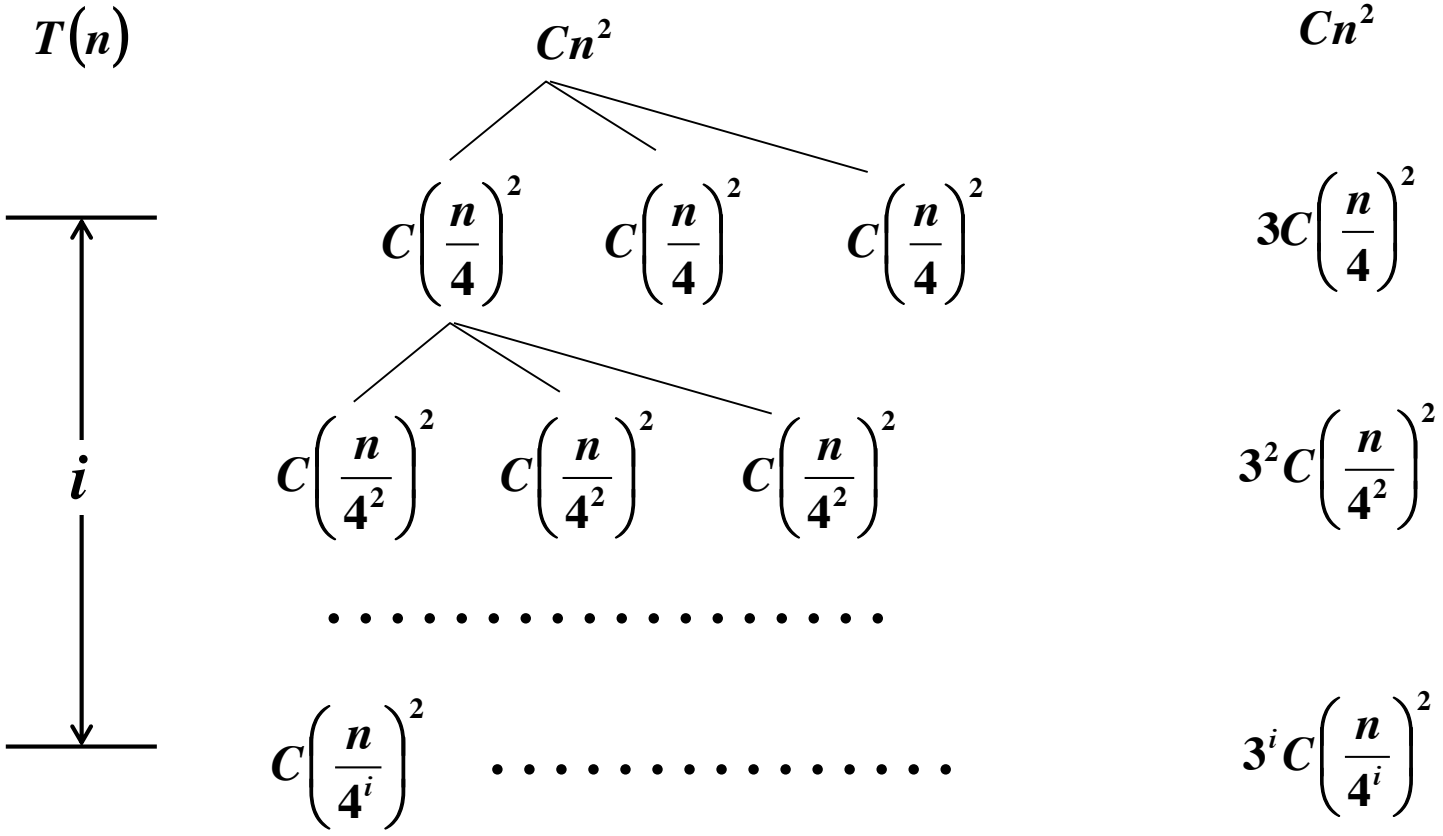
这种方法适用于问题的形式相对简单或比较熟悉，或者问题的解较易猜测的情况。

例1：$T(n)=2T(\lfloor n/2 \rfloor)+n$

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

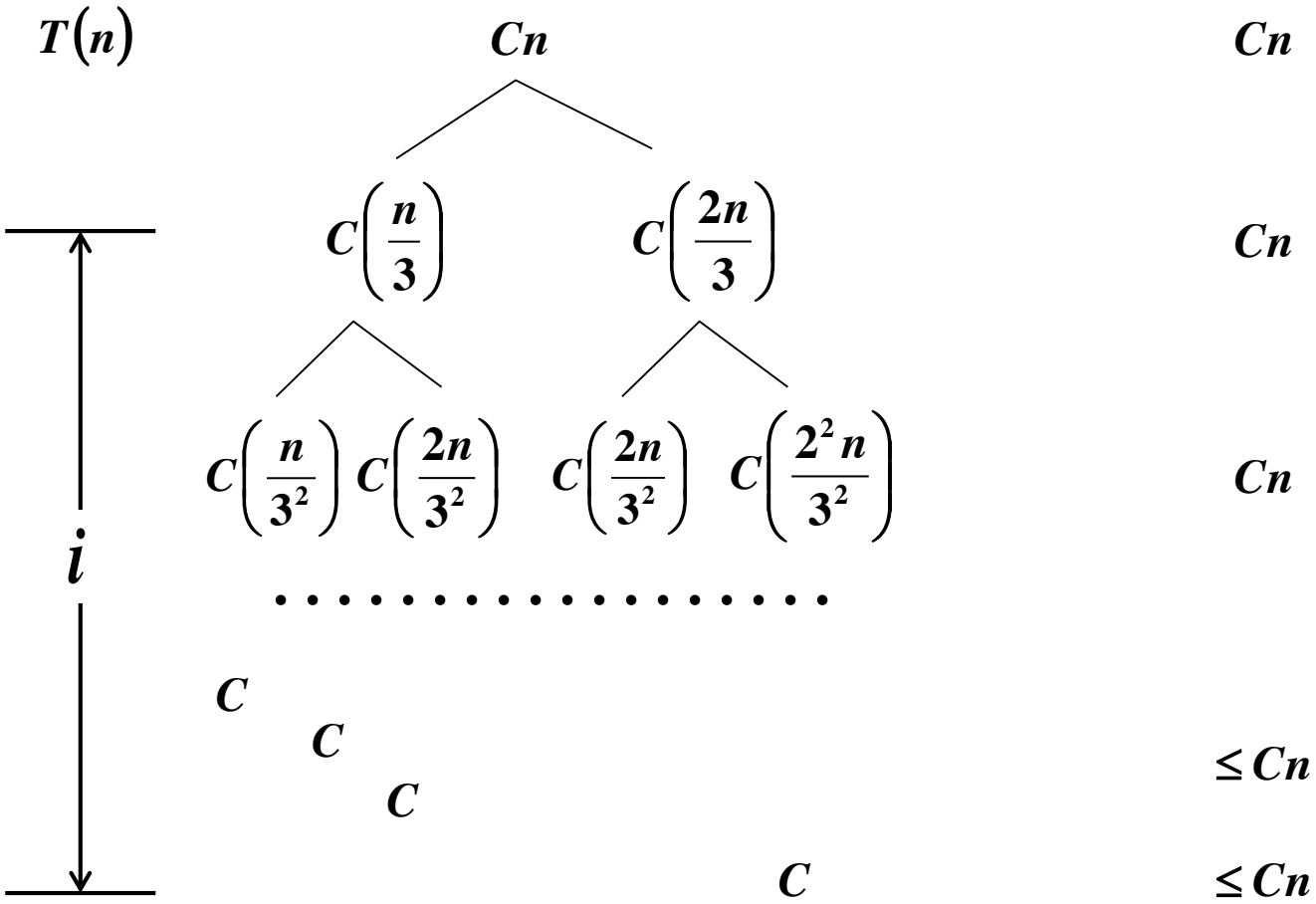
2. 递归树方法(Recursion tree Method)

例3：$T(n)=3T(n/4)+cn^2$



例4: $T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+cn$

解：该问题的子问题规模不一样



3. Master方法

条件：如果T(n)具有如下递归形式：

T(n)=aT(n/b)+f(n)，其中a≥1，b>1且为常数，f(n)为渐进函数

定理4.1 Master定理

对于非负递归函数T(n)=aT(n/b)+f(n)，其中a ≥1，b>1且为常数，f(n)为渐进函数，则T(n)有如下三种渐进界限：

1) if $f(n)=O\left(n^{\log_b a-\varepsilon}\right)$, 常数$\varepsilon>0$, 则$T(n)=\theta\left(n^{\log_b a}\right)$

2) if $f(n)=\theta\left(n^{\log_b a}\right)$, 则$T(n)=\theta\left(n^{\log_b a}\times \lg n\right)$

3) if $f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)$, 常数$\varepsilon>0$且∃常数$c<1$, n足够够大时

有$af\left(\frac{n}{b}\right)\le cf(n)$, 则$T(n)=\theta\left(f(n)\right)$

直观上看，master定理就是比较二个函数f(n)和$n^{\log_b a}$的渐进大小,即：

if $n^{\log_b a} > f(n)$ then T(n)适用条件1)

if $n^{\log_b a} = f(n)$ then T(n)适用条件2)

if $n^{\log_b a} < f(n)$ then T(n)适用条件3)

例5: $T(n) = 9T\left(\frac{n}{3}\right) + n$

例6 : $T(n) = T\left(\frac{2n}{3}\right) + 1$

例7 : $T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$

例8 : $T(n) = 2T\left(\frac{n}{2}\right) + n\lg n$

3.3 快速排序算法设计与分析

1. 快速排序算法设计

是应用分治法的又一例子，实现快速排序算法的基本思想为：

在当前的无序区**A[p..r]**中任取一个元素**x**作为比较的基准，并用该基准将当前无序区分为左右二个较小的无序区**A[p..q-1]**和**A[q+1..r]**，使得左边的无序区**A[p..q-1]**中的元素均小于基准元素**x**，右边的无序区**A[q+1..r]**中的元素均大于**x**。

应用分治法的三个基本步骤求解该问题。

```
Quicksort(A,p,r）
    if p<r
    then q←patition(A,p,r)
          Quicksort(A,p,q-1)
          Quicksort(A,q+1,r)
```

```
Partition(A,p,r)
    x←A[r]
    i←p-1
    for j←p to r-1
    do  if A[j]≤x
            then i←i+1
                    exchange(A[i],A[j])
            exchange(A[i+1],A[r])
    return i+1
```

2. 快速排序算法分析

∵partition算法时间为： $\theta(n)$

∴Quicksort算法时间为：

$T(n) = T(q)+T(n-q-1)+ \theta(n)$

- 最坏情况：

  ∵ 二个无序区的大小分别为n-1和0

  ∴ $T(n) = \theta(n^2)$

- 最好情况：

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

3．平衡划分

假设每次划分都按照9:1的比例划分，则时间为：

$$T(n) \le T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \theta(n)$$

从构造T(n)的递归树上可得递归树的最长路径为：

$$cn \to \frac{9}{10}cn \to \left(\frac{9}{10}\right)^2 cn \to \cdots\cdots \to \left(\frac{9}{10}\right)^i cn$$

令$\left(\frac{9}{10}\right)^i n = 1$，则$n = \left(\frac{10}{9}\right)^i$，$i = \log_{10/9} n$

$$\therefore T(n) \le (i+1)cn = \left(\log_{10/9} n + 1\right)cn = cn\log_{10/9} n + cn = O(n \lg n)$$

4. 随机划分的快速排序算法分析(P91)

```
RANDOMIZED-PARTITION(A,p,r)
    i←RANDOM(p,r)
    exchange(A[r],A[i])
    return PARTITION(A,p,r)

RANDOMIZED-QUICKSORT(A,p,r)
    if p<r
    then q←RANDOMIZED-PARTITION(A,p,r)
         RAMDOMIZED-QUICKSORT(A,p,q-1)
         RAMDOMIZED-QUICKSORT(A,q+1,r)
```

① 最坏情况

$$T(n)=\max_{0\le q\le n-1}\left(T(q)+T(n-q-1)\right)+\Theta(n)$$

猜测$T(n)$有解为$\Theta(n^2)$，替换法可证明$T(n)\le dn^2$

$$\text{proof}: T(n)\le \max_{0\le q\le n-1}\left(dq^2+d(n-q-1)^2\right)+cn$$

$$=d\max_{0\le q\le n-1}\left(q^2+(n-q-1)^2\right)+cn$$

$\because q=0$或$q=n-1$时，$q^2+(n-q-1)^2$具有最大值

$$\therefore T(n)\le d(n-1)^2+cn$$

$$=dn^2-2dn+d+cn$$

$$\le dn^2-2dn+dn+cn \qquad (n\ge 1\text{时})$$

$$=dn^2-dn+cn$$

$$=dn^2-(d-c)n$$

$$\le dn^2 \qquad (\text{取}d\ge c)$$

同理可证$T(n)\ge dn^2$,所以猜测正确$T(n)=\Theta(n^2)$

② 平均情况

定理7.1

令X为排序n个元素时，partition算法所需要执行的总的比较次数，那么快速排序的执行时间为O(n+X)。

求X的二个问题：

1. 任意二个元素什么时候发生比较？
2. 二个元素之间最多比较多少次？

总的比较次数为：

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

二边求期望值得：

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \qquad (\because E[X+Y] = E[X] + E[Y])$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{与} z_j \text{发生了比较}\}$$

$\because z_i$与z_j发生比较只有在z_i或z_j被选为基准元素后

又$\because$集合Z_{ij}中的元素是随机挑选的

$\therefore$每个元素被选为基准的概率相等，为$\dfrac{1}{j-i+1}$

$$\therefore \Pr\{z_i \text{与} z_j \text{发生比较}\}$$

$$= \Pr\{z_i \text{被选为基准}\} + \Pr\{z_j \text{被选为基准}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1}$$

则 $$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{1}{j-i+1} + \frac{1}{j-i+1} \right)$$

$$= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \qquad (\text{令} k = j-i)$$

$$< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k+1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

$$= 2 \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

第四章 红黑树(Red-Black Tree)

4.1 二叉查找树性质(Binary Search Tree)

性质：设x为二叉树中的任一结点，那么对于x左子树中的任一结点y都有key[y]≤key[x],x右子树中的任一结点y都有key[x]≤key[y]。

算法时间：设二叉树高度为h，

静态操作:查找结点、找前驱、后继等时间为O(h)

动态操作：插入和删除结点等时间为O(h)

对于插入操作，新结点总是插入在某个叶结点位置

对于删除操作需依被删结点z的情况而定:

①z无左右孩子

②z只有一个孩子

③z的左右孩子均存在

算法Tree-delete中三个指针x,y,z的含义：

第四章 红黑树(Red-Black Tree)

4.1 二叉查找树性质(Binary Search Tree)

性质：设x为二叉树中的任一结点，那么对于x左子树中的任一结点y都有key[y]≤key[x],x右子树中的任一结点y都有key[x]≤key[y]。

算法时间：设二叉树高度为h，

静态操作:查找结点、找前驱、后继等时间为O(h)

动态操作：插入和删除结点等时间为O(h)

对于插入操作，新结点总是插入在某个叶结点位置

对于删除操作需依被删结点z的情况而定:

①z无左右孩子

②z只有一个孩子

③z的左右孩子均存在

算法Tree-delete中三个指针x,y,z的含义：

4.2 红黑树的特性

红黑树首先是一棵二叉查找树，所以二叉查找树的所有性质红黑树都具有，此外还有自身五个特性：

①每个结点要么是黑色要么是红色

②树根结点的颜色为黑色

③叶结点(nil)为黑色

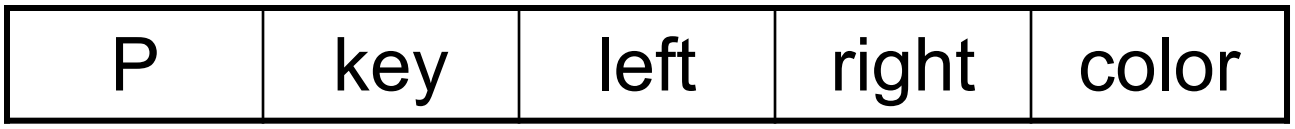
④如果某个结点为红色，则它的左、右孩子结点均为黑色

⑤对树中任一结点，所有从该结点出发到其叶结点的路径中均包含相同数目的黑色结点

外部结点(external node):不包含实际关键字

内部结点(internal node):包含实际关键字

红黑树结点的形式为：



图例见书P164

4.3 黑高度(Black height)

def: 黑高度bh(x)表示从x结点出发(不包含x结点)到其叶结点路径上的黑色结点个数。

定理13.1

具有n个内部结点的红黑树高度最多为2lg(n+1)。

proof: 设红黑树的高度为h，即要证明$h \le 2\lg(n+1)$

$$\Rightarrow h/2 \le \lg(n+1)$$

$$\Rightarrow 2^{h/2} \le n+1$$

$$\Rightarrow 2^{h/2} - 1 \le n$$

令树的黑高度为bh,根据红黑树性质4,5可知，

从根结点出发的任一路径上的黑色结点数 $\ge h/2$

则：$bh \ge h/2 \Rightarrow 2^{bh} - 1 \ge 2^{h/2} - 1$

如果$2^{h/2}-1 \le 2^{bh}-1 \le n$ 成立，则定理得证

用归纳法证明对于树中任一结点x为根的子树至少

包含$2^{bh(x)}-1$个内部结点

归纳基础：当$bh(x)=0$时，说明此时x必为叶结点

又 $\because 2^{bh(x)}-1=2^0-1=0$

$\therefore$归纳基础成立

归纳假设：假设黑高度为$bh(x)-1$时命题成立

归纳步骤：$\because$ x为根的子树内部结点数＝x左子树的内部结点数＋

x右子树的内部结点数＋1

$$\ge \left(2^{bh(x)-1}-1\right)+\left(2^{bh(x)-1}-1\right)+1$$

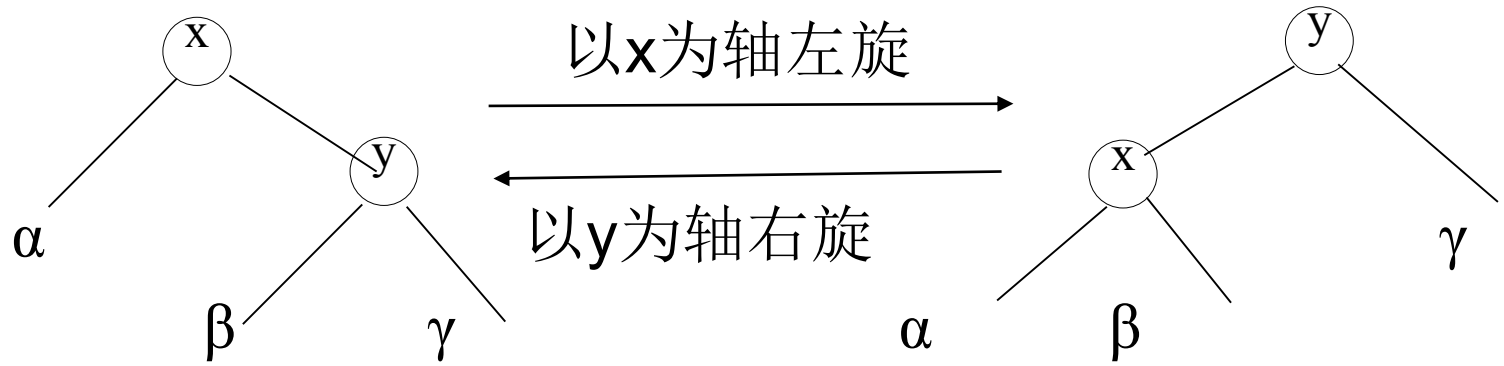
$$=2^{bh(x)}-1$$

特别地，当x为树根结点时，则红黑树至少包含$2^{bh}-1$个内部结点

$\therefore n \ge$ x为根的子树内部结点数 $\ge 2^{bh}-1 \ge 2^{h/2}-1$，定理得证

4.4 红黑树的调整操作

旋转(rotate)分为两种：左旋和右旋操作



左旋算法如下:(P166)

left-rotate(T,x)

```
    y←right[x]
    right[x]←left[x]
    p[left[y]]←x
    p[y]←p[x]
    if p[x]=nil
    then root[T]←y
    else if x=left[p[x]]
            then left[p[x]]←y
            else right[p[x]]←y
    left[y]←x
    p[x]←y
```

4.5 红黑树的插入操作

插入一个新结点分为二步完成：

1. 按照二叉查找树的方式插入新结点z
2. 恢复红黑树的特性

算法如下：

```
RB-Insert(T,z)
    Tree-Insert(T,z)
    left[z]←nil
    right[z]←nil
    color[z]←red
    RB-Insert-fixup(T,z)
```

首先检查红黑树5个特性可能遭受破坏的情况，然后针对遭受破坏的特性制定具体恢复的策略。

```
RB-Insert-fixup(T,z)   (P167)
    while color[p[z]]=red
       do  if  p[z]=left[p[p[z]]]
             then y←right[p[p[z]]]
                  if color[y]=red
                  then color[p[z]]←black                          case1
                       color[y]←black                             case1
                       color[p[p[z]]]←red                         case1
                       z←p[p[z]]                                  case1
                  else if z=right[p[z]]
                       then z←p[z]                                case2
                            left-rotate(T,z)                      case2
                       color[p[z]]←black                          case3
                       color[p[p[z]]]←red                         case3
                       right-rotate(T,p[p[z]]])          case3
             else      same as “then clause”
    color[root[T]]←black
```

RB-delete-fixup算法具体调整过程为：

case1: if z的uncle结点y=right[p[p[z]]]为红色

op: color[p[z]]←black

color[y]←black

color[p[p[z]]]←red

z=p[p[z]]

case2: if y为黑色且z是其父结点的右孩子

op: z=p[z]

left-rotate(T,z)

case3: if y为黑色且z是其父结点的左孩子

op: color[p[z]]←black

color[p[p[z]]←red

right-rotate(T,p[p[z]])

退出while循环后，根结点为红色的两种情况：

1. 新结点插入到空树中；

2. 新结点的uncle结点为红色，且多次变换后uncle结点总为红色

例：构造初始为空树，依次插入关键字8，19，12，31，38，35后的红黑树

RB-Insert算法分析：

∵ 红黑树的高度h=O(lgn)

Tree_Insert算法时间为：O(lgn)

RB-Insert-fixup算法时间为：O(lgn)

∴RB-Insert算法时间为：O(lgn)

4.6 红黑树的删除操作

与插入操作相似，删除一个结点也分为二步完成：

1. 按照二叉查找树的方式删除结点
2. 恢复红黑树的特性

算法如下：(P173)

```
RB-delete(T,z)
    Tree-delete(T,z)
    if color[y]=black
    then RB-delete-fixup(T,x)
    return y
```

RB-delete-fixup入口分析：

1. if z无左右孩子且为黑色，then 特性⑤破坏
2. if z只有一个孩子且为黑色，then 特性⑤破坏、特性④可能破坏
3. if z的左右孩子均存在，then z的中序后续y的情况等同与1,2

```
RB-delete-fixup(T,x)
    while x≠root[T] and color[x]=black
        do if x=left[p[x]]
            then w←right[p[x]]
                    if color[w]=red
                    then color[w]←black                                  case1
                           color[p[x]]←red                               case1
                           left-rotate(T,p[x]])                          case1
                          w←right[p[x]]                                  case1
                    if color[left[w]]=black and color[right[w]]=black
                    then color[w]←red                          case2
                          x←p[x]                                         case2
                    else if color[right[w]]=black
                          then color[left[w]]←black                      case3
                                color[w]←red                             case3
                                right-rotate(T,w)                        case3
                                w←right[p[x]]                            case3
                           color[w]←color[p[x]]                          case4
                           color[p[x]]←black                             case4
                           color[right[w]]←black                         case4
                           left-rotate(T,p[x]])                          case4
                          x←root[T]
            else same as “then clause”
    color[x]←black
```

算法RB-delete-fixup具体调整过程

case1: x的兄弟结点w=right[p[x]]为红色

op: color[w]=black
color[p[x]]=red
left-rotate(T,p[x])
w=right[p[x]]

case2: w及w的左右孩子均为黑色

op: color[w]＝red
x=p[x]

case3: w为黑色，w的左孩子为红色、w的右孩子为黑色

op: color[left[w]]=black
color[w]=red
right-rotate(T,w)
w=right[p[x]])

case4: w为黑色，w的右孩子为红色

op:　color[w]=color[p[x]]

　　color[p[x]]=black

　　color[right[w]]=black

　　left-rotate(T,p[x])

RB-delete算法时间：

∵ 红黑树的高度h=O(lgn)

　Tree-delete算法时间为O(lgn)

　RB-delete-fixup算法时间为O(lgn)

∴ RB-delete算法时间为O(lgn)

第五章 数据结构扩张(Augmenting Data Structures)

5.1 动态序统计(Dynamic Order Statistics)

所谓序统计就是在一系列数中找出最大、最小值，某个数的序值等操作。

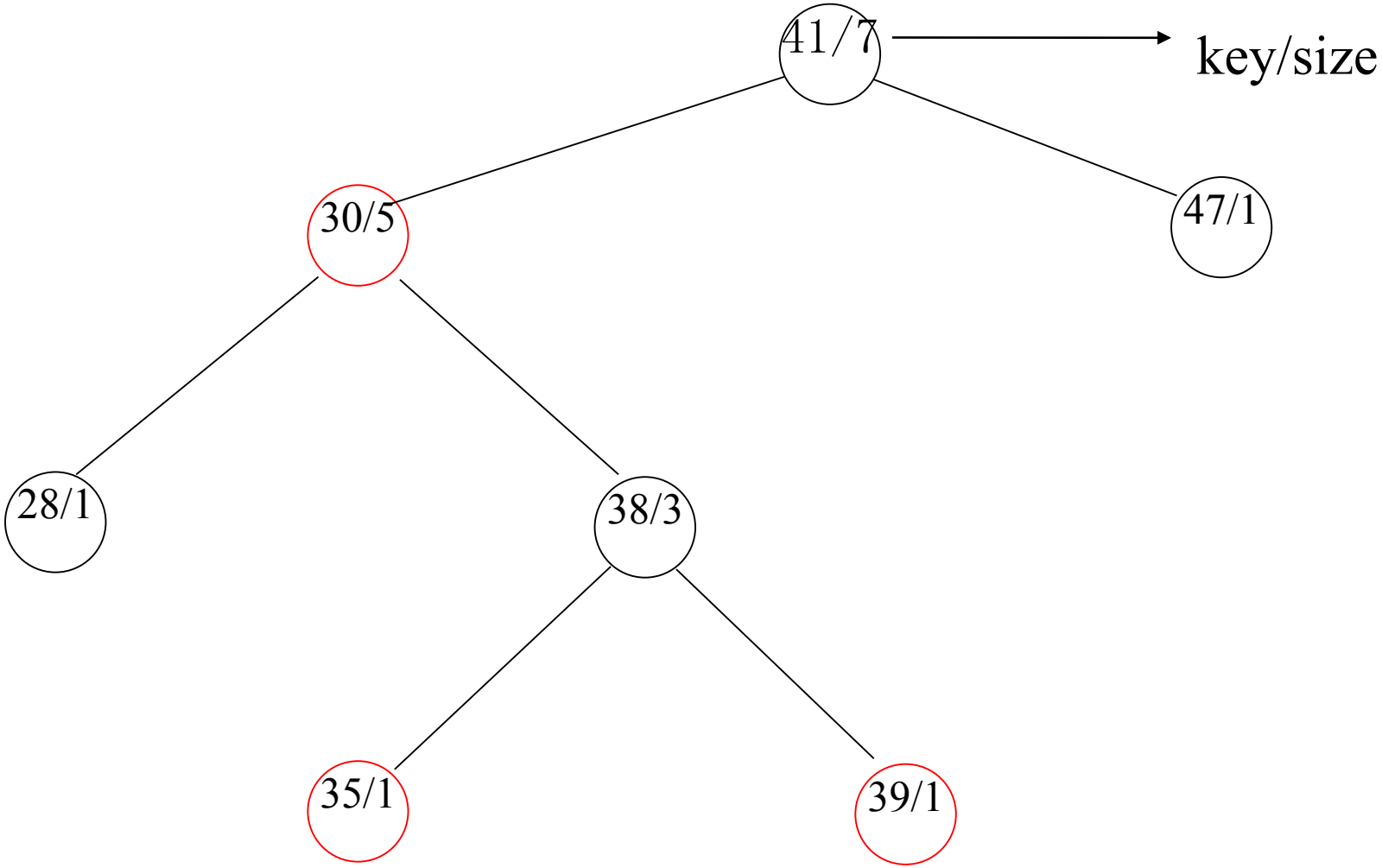
通过对红黑树扩充相应域后可得到解决动态序统计问题的序统计树。序统计树结点的形式为：

P	left	right	key	color	size

Size[x]域的定义为：以x为根的子树所包含的内部结点数(包括x本身)。即：

if 外部结点(nil)的size值定义为0，then 对任意的x有:

size[x]=size[left[x]]+size[right[x]]+1


41/7
key/size
30/5
47/1
28/1
38/3
35/1
39/1

1. 找第i个最小值操作(P182)

```
OS-select(x,i)          //找x为根结点的第i个最小数
    r←size[left[x]]+1
    if  i=r
    then   return x
    else   if i<r
               then return OS-select(left[x],i)
               else return OS-select(right[x],i-r)
```

算法时间：O(lgn)

2. 求x的序值操作 (P182)

```
OS-rank(T,x)
    r←size[left[x]]+1
    y←x
    while y≠root[T]
        do if  y=right[p[y]]
            then r←r+size[left[p[y]]]+1
            y←p[y]
    return r
```

算法时间：O(lgn)

3. 插入一个结点的size域调整

调整分为二步完成，

第一步 新结点插入到树中对size域的影响并调整之

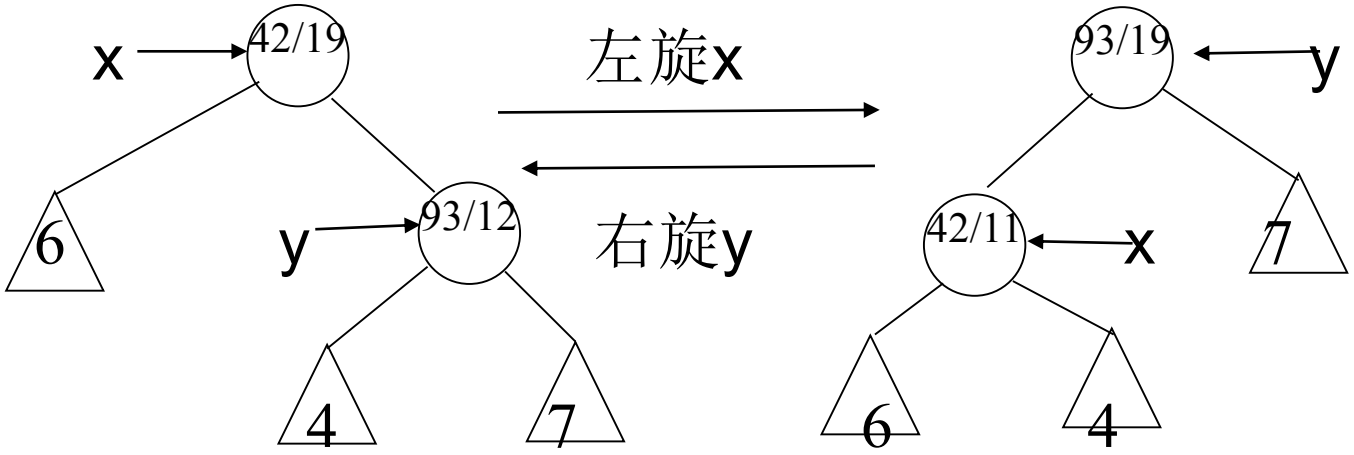
在这一步中受影响的结点是新插入结点的一系列祖先结点，所以只要对这一系列祖先结点的size域值加1即可。

第二步 颜色调整过程中对size域的影响并调整之

在这一步中只有旋转操作将影响结点的size值，此时调整的方法为（假设为左旋操作）：

size[y]=size[x]

size[x]=size[left[x]]+size[right[x]]+1



4. 删除一个结点size域的调整

同样删除一个结点也分为二步完成：

第一步 删除一个结点将使被删结点的size受到影响，调整时所有受影响结点的size域值减1即可

第二步 颜色调整只有旋转操作影响结点的size域，调整方法同插入一个结点。

5. 插入和删除一个结点的时间

∵ 在插入和删除一个结点的第一步中，size域受影响的结点最多是从根结点到叶结点的单条路径

又∵红黑树的高度为O(lgn)

∴ 这一步的size值调整时间为O(lgn)

∵在插入和删除一个结点的第二步中，一次旋转操作size域值的调整时间为O(1)且旋转操作最多只有三次

∴这一步的size值调整时间为O(1)

因此在序统计树上完成一次插入或删除操作的时间为O(lgn)

5.1 数据结构扩张的步骤

1. 挑选一个合适的基本数据结构
2. 决定在基本数据结构上应增加的信息
3. 修改基本数据结构上的操作并维持原有的性能
4. 修改或设计新的操作

定理14.1 (关于在红黑树上增加信息的条件）

令f为具有n个结点的红黑树扩张后的一个域，如果结点x的f域值仅需通过结点x、left[x]、right[x]以及f(left[x])、f(right[x])就可获得，那么插入和删除操作对树T中所有结点f值进行维护将维持O(lgn)的性能。

proof:

∵ 结点x的f值只和x自身及x的左右孩子有关

∴ 某个结点的f值发生变化，受影响的仅是该结点的一系列的祖先结点

∵ 红黑树的高度为O(lgn)

∴ 需要调整f值的结点数最多为O(lgn)个

又∵旋转操作最多执行三次

∴ 在插入和删除的颜色调整算法中结点f值的调整时间为O(1)

因此f值的调整时间最多为O(lgn)，将维持红黑树O(lgn)性能。

5.3 区间树(interval tree)

区间定义：

闭区间(closed interval)：存在一实数对$[t_1,t_2]$，满足

$$\{t \in R: t_1 \le t \le t_2\}$$

开区间：$(t_1，t_2)$

半开区间：$[t_1，t_2)$、$(t_1，t_2]$

扩张步骤：

1. 挑选红黑树作为基本数据结构
2. 确定需增加的信息域

区间的三分原则：（令i和i’为二个区间）

① i在i’的左边

② i在i’的右边

③ i与i’重叠

其中重叠(overlap)的定义为：$i \cap i' \neq \Phi$

如果令low[i]=t_1，high[i]=t_2，

那么i与i’重叠可描述为：

low[i]≤high[i’]

low[i’] ≤high[i]

区间比较的关系见图（P187）

确定建树和新增域：以区间的低端点作为建树的比较关键字，区间的高端点作为新增的域，即：

max[x]=Max{high[int[x]，max[left[x]]，max[right[x]]}

其中int[x]表示区间x，max为新增的域名

原红黑树结点新增域max后扩展为区间树，如图（P187）所示。

3. 核实增加max域后是否可以维持红黑树原有的性能

①旋转操作：以左旋为例，调整如下：

max[y]=max[x]

max[x]=Max{high[int[x]，max[left[x]]，max[right[x]]}

所以旋转调整时间仍为O(1)。

②插入和删除操作：

∵max域的值只和x的高端点、x左右孩子的max域有关

∴由定理14.1可知，区间树的插入和删除操作仍维持O(lgn)的性能

4. 新增区间树的查找操作

Interval-search(T,i)　//在树T中查找与i重叠的区间

x←root[T]

while x≠nil and i与int[x]不重叠

do if left[x]≠nil and max[left[x]]≥low[i]

then　x←left[x]

else　x←right[x]

return x

算法时间：容易看出，查找路径最多是从根到某个叶结点的单条路径。

∵红黑树的高度为O(lgn)

∴Interval-search算法的时间为O(lgn)

定理14.2

算法Interval-search(T,i)要么返回与i重叠的某个结点，要么返回为空表示树T中未找到与i重叠的区间。

proof：定义循环不变式为：

只要树T中存在与i重叠的区间，则x指向该结点

初始状态：∵进入循环前，x=root[T]

∴此时x与i重叠，则x指向该区间

保持状态：while循环体存在二个分支x←left[x]和x←right[x],分别讨论这两种情况

①if执行了分支x←right[x]，说明条件left[x]=nil或者max[left[x]]<low[i]成立

如果由条件left[x]=nil进入，说明x的左子树为空，那么只要x的右子树才可能存在与i重叠的区间

∴设置x←right[x]正确

如果由条件max[left[x]]<low[i]进入，

令i’为x左子树中的任一区间，根据区间树的性质可知：

high[i’]≤max[left[x]]<low[i]

即：high[i’]<low[i]

说明x的左子树中的任一区间都不会与i重叠

∴设置x←right[x]正确

②if 执行分子x←left[x]，说明条件left[x]≠nil同时max[left[x]]≥low[i]成立

假设在x的左子树中不存在与i重叠的区间并令i’为x左子树中的任一区间，根据区间不重叠的条件有：

high[i’]<low[i]

或 high[i]<low[i’] 成立

∵进入分支x←left[x]的条件为max[left[x]]≥low[i]，根据max域的定义，在x的左子树中一定存在一个区间i''，使得 high[i'']=max[left[x]] ≥low[i]，这与不重叠条件high[i']<low[i]矛盾

∴i与i'不重叠的条件只可能是：high[i]<low[i']

令i'''为x右子树中的任一区间，由区间树的性质可得：

low[i''']≥low[i']>high[i]

即： low[i''']>high[i]

∴此时x的右子树中也不存在与i重叠的区间

∴设置x←left[x]正确

终止状态：退出循环时，x要么指向某个重叠区间要么x=nil表示树T中没有与i重叠的区间。

定理得证

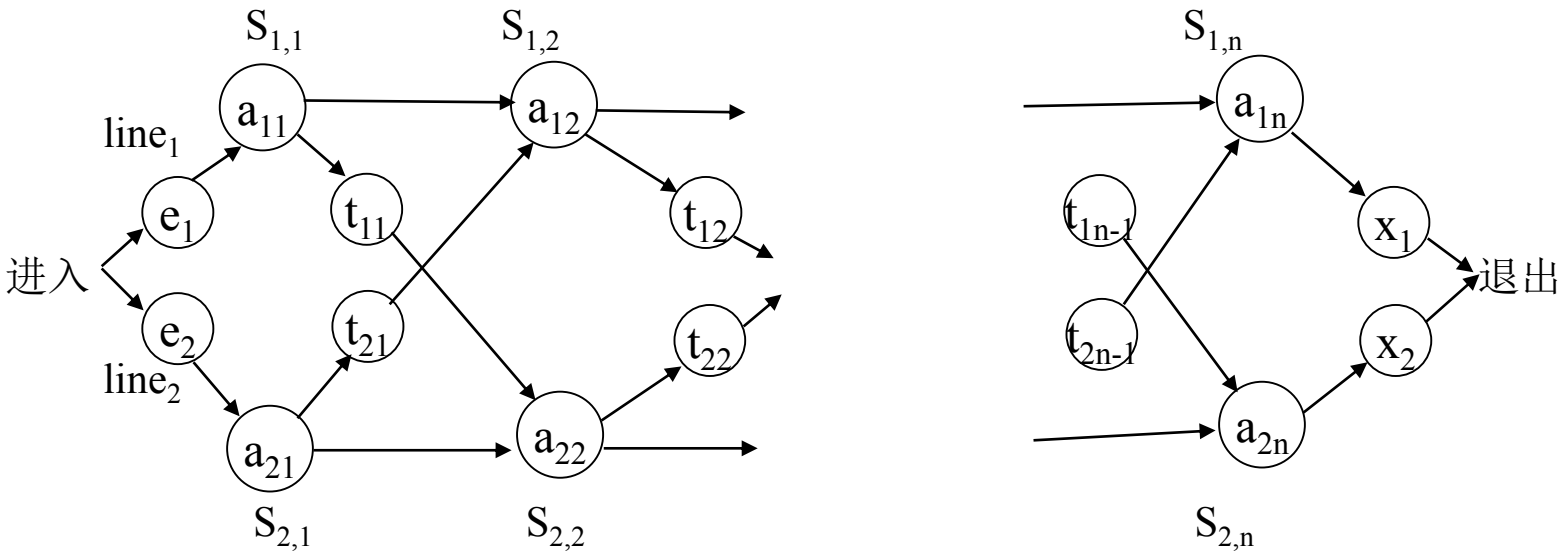
第六章 动态规划(Dynamic Programming)

6.1 动态规划方法的基本步骤

1. 描述问题的最优解(optimal solution)结构特征
2. 递归定义最优解值
3. 自底向上计算最优解值
4. 从已计算得到的最优解值信息中构造最优解

6.2 装配线调度问题(Assembly-line Scheduling)

1. 问题的提出



其中：$S_{i,j}$表示第i条装配线中第j个装配站，i=1,2,…..,n

$a_{i,j}$表示在第i条装配线、第j个装配站装配某个部件所需的时间

$t_{i,j}$表示从装配线i、装配站j切换到装配线j所需的时间

e_i表示进入装配线i所需的时间

x_i表示退出装配线i所需的时间

问题是如何找一条从开始到退出的最快路线？（如图15-2，P193）

2. 枚举法求解

3. 动态规划方法求解

step1: 描述问题的最优解结构特征

观察从开始到某个装配站$S_{1,j}$的最优路线

①if j=1 then 从开始到$S_{1,1}$只有一条路线

②if j=2 then 从开始到$S_{1,2}$有两条路线，一条是$S_{1,1}$直接到达$S_{1,2}$，另一条是从$S_{2,1}$切换到达$S_{1,2}$。

同理可知，从开始到$S_{1,j}$也只有两条路线，一条是从$S_{1,j-1}$直接到达，另一条是从$S_{2,j-1}$切换到达。

那么从开始到达$S_{1,j}$的最优解结构特征是什么呢？

分析如下：假设从开始到$S_{1,j}$的一条最优路线是从开始到$S_{1,j-1}$再直接到达$S_{1,j}$的，那么子路径P_1从开始到$S_{1,j-1}$也一定是最优的。

proof:（采用cut and paste方法证明）

∵如果子路径P_1从开始到$S_{1,j-1}$不是最优的，那么一定存在一条从开始到$S_{1,j-1}$的更优子路径P_2，当用P_2去替换原子路径P_1后，将得到一条比原路径更优的路线，这与假设从开始到$S_{1,j}$是一条最优路线矛盾。

∴子路径P_1一定也是最优的

同理可知，若从开始到$S_{1,j}$的一条最优路线是从开始到$S_{2,j-1}$再切换到达$S_{1,j}$的，那么子路径从开始到$S_{2,j-1}$也一定是最优的。

以上这种最优解结构特征有称为最优子结构(optimal substructure)性质，即如果问题的解是最优的，则所有子问题的解也是最优的。

step2：递归定义最优路线的最快时间

令$f_i[j]$为经过装配站$S_{i,j}$的最快时间

则$f_1[n]$表示经过装配站$S_{1,n}$的最快时间

则$f_2[n]$表示经过装配站$S_{2,n}$的最快时间

∴从开始进入到退出的最快时间为：

$f^*=\min(f_1[n]+x_1，f_2[n]+x_2)$

要得到f*值，需要计算$f_i[j]$的每个值

$f_1[j]=\min\{f_1[j-1]+a_{1,j}，f_2[j-1]+t_{2,j-1}+a_{1,j}\}$

$f_2[j]=\min\{f_2[j-1]+a_{2,j}，f_1[j-1]+t_{1,j-1}+a_{2,j}\}$

递归初始值为：

$f_1[1]=e_1+a_{1,1}$

$f_2[1]=e_2+a_{2,1}$

step3：自底向上计算最快时间

先确定此问题的底，然后再自底向上计算最优解值

算法Fastest-Way见书P196。

容易看出该算法的时间为$\Theta(n)$。

step4：构造最优解结构（输出最优路线）

算法Print-Station见书P196。

6.3 矩阵链乘(Matrix-chain Multiplication)

1. 问题的提出

给定一个矩阵序列$<A_1,A_2,\ldots,A_n>$，其中矩阵A_i的维数为$P_{i-1}\times P_i$，要求计算$A_1\times A_2\times\cdots\times A_n$矩阵链乘的乘法次数最少？

例：四个矩阵相乘$A_1\times A_2\times A_3\times A_4$的链乘顺序有：

$(A_1(A_2(A_3A_4)))$　　又称为矩阵完全加括号形式

$(A_1((A_2A_3)A_4))$　　fully parenthesized

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$

共有五种链乘顺序

例：假设有三个矩阵$A_1 \times A_2 \times A_3$，其中矩阵的维数为：$10 \times 100$，$100 \times 5$，$5 \times 50$。

对于矩阵链乘问题首先检查枚举法是否可行？

2. 动态规划方法求解此问题

step1:问题的最优子结构

假设子问题A_{ij}的解$(A_iA_{i+1}\ldots A_j)$是一个最优解，则在A_{ij}中一定存在一个最佳分裂点$k(i \le k < j)$，使得子链A_{ik}和A_{k+1j}的解$(A_iA_{i+1}\ldots A_k)$和$(A_{k+1}\ldots A_j)$也是最优的。

proof: cut and paste方法证明

假设子链$A_{ik}=(A_iA_{i+1}\ldots A_k)$不是最优的完全加括号形式，则一定存在一种更优的完全加括号形式$A_{ik}'=(A_iA_{i+1}\ldots A_k)$，使得$A_{ik}'$的乘法次数比$A_{ik}$更少，即$|A_{ik}'|<|A_{ik}|$，那么当我们用$A_{ik}'$去替换$A_{ik}$后将得到一个比原问题$A_{ij}$更少乘法的完全加括号形式。即：

原最优解$A_{ij}=((A_{ik})(A_{k+1j}))$，

乘法次数为$|A_{ij}|=|A_{ik}|+|A_{k+1j}|+P_{i-1}\times P_k\times P_j$

新的解$A_{ij}'=((A_{ik}')(A_{k+1j}))$，

乘法次数为$|A_{ij}'|=|A_{ik}'|+|A_{k+1j}|+P_{i-1}\times P_k\times P_j$

$\because |A_{ik}'|<|A_{ik}|$

$\therefore |A_{ij}'|<|A_{ij}|$，这与$A_{ij}$最优是矛盾的

$\therefore$子链A_{ik}一定是最优的

同理可证子链A_{k+1j}也一定是最优的

step2：递归定义最优解值

令m[i,j]存放子链$A_{ij}=(A_iA_{i+1}\ldots A_j)$的乘法次数

则m[1,n]表示所有n个矩阵链乘的乘法次数

∵if i=j then 只有一个矩阵

∴ m[i,i]=0，i=1,2….n

∵if i<j then 根据矩阵链乘的最优子结构性质可知，此时存在一个最佳分裂点k，使得A_{ij}可分裂为二个子链A_{ik}和A_{k+1j}

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

∵原问题是求最少的乘法次数

$$\therefore m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\left(m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\right) & i<j\end{cases}$$

step3: 自底向上计算最优解值

```
matrix-chain-order(P)
    n←length[p]-1
    for  i←1 to n
     do  m[i,i]←0
    for l=2 to n                          //l为链长
     do for  i←1 to n-l+1                 //具有n-l+1个链长为l的组合
          do   j←i+l-1
               m[i,j]←∞
               for   k←i to j-1           //找最佳分裂点k
                do   q←m[i,k]+m[k+1,j]+P_{i-1}P_kP_j
                     if q<m[i,j]
                     then m[i,j]←q
                          s[i,j]←k        //记录最佳分裂点
    return m and s
```

算法时间分析：

∵算法matrix-chain-order包含三重循环

又∵每重循环的次数≤n

∴算法时间为$O(n^3)$。

step4: 输出最优解结构

算法见书P201

根据S数组记录的最佳分裂点k值可构造出n个矩阵链乘的最优加括号形式

如书P200，图15－3所示。

6.4 动态规划的要素

1. 最优子结构性质

最优子结构性质是应用动态规划方法的必要前提，即所解问题必须具有最优子结构性质才能用动态规划方法求解。

所谓最优子结构性质是指一个问题的最优解中所包含的所有子问题的解都是最优的。

对于一个问题发现其最优子结构性质的几个因素：

①检查问题的解所面临的选择

②假定某种选择为一最优解

③分析这种选择将产生多少子问题

④证明子问题的解也是最优的

最优子结构的细节问题：

特别需要指出的是当问题本身不具有最优子结构性质时不能滥用最优子结构性质。

例如：对于一个有向图**G=(V,E)**

. 如果问题是找图G中顶点u, v∈**V**的最短路径

假设路径**P**是从**u**到**v**的一条最短路径，

即**P**：$u \xrightarrow{P_1} s \xrightarrow{P_2} v$

∵只要**P**是最短路径，则子路径P_1和P_2也一定是最短路径

∴ 该问题具有最优子结构性质

.如果是找二个顶点**u,v** ∈ **V**间的最长路径，那么该问题是否具有最优子结构性质呢？

2. 重叠子问题

虽然问题分解后是否存在重叠子问题不是应用动态规划方法的必要前提，但存在重叠子问题应用动态规划方法可以提高算法效率。

```
Recursive-matrix-chain(P,i,j)
    if i=j  then return 0
    m[i,j]←∞
    for k←i to j
     do q←Recursive-matrix-chain(P,i,k)
          +Recursive-matrix-chain(P,k+1,j)+P_{i-1}P_kP_j
        if q<m[i,j]
        then m[i,j]←q
    return m[i,j]
```

算法时间分析：

$$T(1) \geq 1 \qquad \text{if } n = 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1 \right) \qquad \text{if } n > 1$$

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

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

$$= n + 2\sum_{k=1}^{n-1} T(k)$$

用替换法猜测T(n)有解为$\Omega(2^{n-1})$

proof：归纳基础：$\because n=1$时，$2^{n-1}=1$

$$\therefore T(1)=1$$

归纳假设：$T(k)\geq 2^{k-1}$

归纳步骤：$T(n)\geq n+2\sum_{k=1}^{n-1}2^{k-1}$

$$=n+2(2^{n-1}-1)$$

$$\geq 2^{n-1}$$

$$\therefore T(n)=\Omega\left(2^{n-1}\right)$$

3. 记忆法(Memorization)

记忆法是动态规划方法的一种变种形式，它采用自顶向下计算最优解值的方法。

Lookup-chain(P,i,j)

if m[i,j]<∞ then return m[i,j]

if i=j

then m[i,j]←0

else for k←i to j-1

do q←Lookup-chain(P,i,k)

+Lookup-chain(P,k+1,j)+$P_{i-1}P_kP_j$

if q<m[i,j] then m[i,j]←q

return m[i,j]

6.5 最长公共子序列(Longest Common Subsequece)问题

1. 子序列

令给定序列$X=\{x_1,x_2,\ldots x_m\}$，另一序列$Z=\{z_1,z_2,\ldots z_k\}$是X的子序列必须满足：X的下标中存在一个严格的递增序$i_1<i_2<\ldots<i_k$，使得对所有的j均有$Xi_j=Z_j(1\le j\le k)$。

例：$X=\{A,B,C,B,D,A,B\}$，序列$Z=\{B,C,D,B\}$是X的一个子序列，存在下标序列{2,3,5,7}。

2. 公共子序列

对于序列X和Y，序列Z如果既是X的子序列又是Y的子序列，则Z是X和Y的公共子序列。

例： $X=\{A,B,C,B,D,A,B\}$

$Y=\{B,D,C,A,B,A\}$

则$Z=\{B,C,A\}$是X和Y的公共子序列

3.最长公共子序列(LCS)

子序列{B,C,B,A}、{B,D,A,B}是X和Y的LCS。

4. 动态规划方法求解LCS问题

step1: 找出LCS问题最优子结构性质

定理15.1 （P351）

令$X=\{x_1,x_2,\ldots x_m\}$，$Y=\{y_1,y_2,\ldots y_n\}$为二个序列，子序列$Z=\{z_1,z_2,\ldots z_k\}$是X和Y的一个最长公共子序列，则：

①if $x_m=y_n$ then $z_k=x_m=y_n$且Z_{k-1}是X_{m-1}和Y_{n-1}的LCS

②if $x_m\neq y_n$且$z_k\neq x_m$ then Z是X_{m-1}和Y的LCS

③if $x_m\neq y_n$且$z_k\neq y_n$ then Z是X和Y_{n-1}的LCS

proof:

①if $z_k \neq x_m$,then将存在一个长度大于k的LCS$\{z_1,z_2 \ldots z_k,x_m\}$，这与Z是X和Y的LCS矛盾

$\therefore z_k=x_m=y_n$

又∵若存在一个长度大于k-1的X_{m-1}和Y_{n-1}的LCS，那么添加x_m后长度将大于k，这与Z是X和Y的LCS矛盾

∴Z_{k-1}是X_{m-1}和Y_{n-1}的LCS

②if 存在一个长度大于k的X_{m-1}和Y的LCS，那么该LCS也应是X和Y的LCS且长度大于k，这与Z是X和Y的LCS矛盾

③证明同②

step2: 递归定义LCS值

令C[i,j]存放子序列X_i和Y_j的LCS长度

∵i=0或j=0时，表示X_0或Y_0为空串

∴C[0,j]=C[i,0]=0

其他情况：

$$C[i,j]=\begin{cases}0 & i \text{ or } j=0\\ C[i-1,j-1]+1 & i,j>0 \text{ and } x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & i,j>0 \text{ and } x_i\neq y_j\end{cases}$$

step3: 自底向上计算LCS值

算法LCS-length(X,Y)见书P210

算法时间为Θ(mn)

6.6 最优二叉查找树(Optimal Binary Search Tree)

1. 问题的提出

对于单个关键字的查找，在红黑树上查找时间为O(lgn)，但若查找的是一系列关键字且每个关键字的查找频度值不同，此时从整体来看红黑树不能产生最少的时间

令n个不同的关键字集$K=\{k_1,k_2,...,k_n\}$，其中$k_1<k_2<\ldots<k_n$

p_i：检索关键字k_i的概率

令$d_0,d_1,\ldots,d_n$表示不在关键字集K中的虚拟(dummy)关键字，其中：

d_0：小于k_1的所有关键字

d_n：大于k_n的所有关键字

d_i：介于k_i和k_{i+1}之间的所有关键字(i=1,2,…,n-1)

q_i：检索关键字d_i的概率

例：书P213

对书中的关键字检索只有二种状态，即：

成功检索：找到关键字k_i，概率为p_i

不成功检索：找到关键字d_i，概率为q_i

$\therefore$ 概率和 $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$

如果定义查找关键字k_i的代价(比较次数)为从根结点到k_i结点的比较次数，则树T的平均检索代价为：

$$E[T] = \sum_{i=1}^{n} (depth(k_i) + 1) \times p_i + \sum_{i=0}^{n} (depth(d_i) + 1) \times q_i$$

$$= 1 + \sum_{i=1}^{n} depth(k_i) \times p_i + \sum_{i=0}^{n} depth(d_i) \times q_i$$

具有最小E[T]值的二叉查找树就称为最优二叉查找树

step1：OBST的最优子结构性质

如果T_r是一棵以k_r为根且包含$k_i,\ldots,\ldots,k_j$的OBST，那么包含关键字$k_i,\ldots k_{r-1}$的左子树T_l及包含关键字$k_{r+1},\ldots,k_j$的右子树T_r也是OBST。

proof: 如果左子树T_l不是最优的，则一定存在一棵更优的包含关键字$k_i,\ldots,\ldots,k_{r-1}$的二叉查找树$T_l'$，使得：

$E[T_l']<E[T_l]$，那么用T_l'去替换原左子树T_l，则得到一棵比T_r更优的树，这与T_r是最优的矛盾。

∴左子树T_l也是最优的

同理可证右子树T_r也是OBST。

step2：递归定义最优解值

令e[i,j]为包含关键字$k_i,\ldots,k_j$的OBST平均检索代价，则e[1,n]为所求解，其中$i\geq 1$，$j\leq n$且$j\geq i-1$

令$k_r(i \le r \le j)$为该子树的根结点，则：

$k_i,\ldots,k_{r-1}$为T_r的左子树，同时包含$d_{i-1},\ldots d_{r-1}$个虚拟关键字

$k_{r+1},\ldots,k_j$为T_r的右子树，同时包含$d_r,\ldots,d_j$个虚拟关键字

特别地：r=i时，左子树$ki,\ldots,k_{i-1}$不包含实际关键字，仅包含一个虚拟关键字d_{i-1}

同样地：r=j时，右子树$k_{j+1},\ldots,k_j$不包含实际关键字，仅包含一个虚拟关键字d_j

令全函数$w(i,j)=\sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$，则

$$e[i,j] = p_r + e[i,r-1] + w(i,r-1) + e(r+1,j) + w(r+1,j)$$
$$= w(i,j) + e[i,r-1] + e[r+1,j]$$

特别地：

当$j = i-1$时，子树$k_i,\cdots,k_j$只包含

一个虚拟关键字d_{i-1}

$\therefore w(i,i-1) = q_{i-1}$

$e[i,i-1] = q_{i-1}$

∵求e[i,j]的最小值

$$\therefore e[i,j] = \begin{cases} q_{i-1} & j = i-1 \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & j \ge i \end{cases}$$

step3：计算最优解值

算法见书

算法时间为$O(n^3)$

第七章 贪心法(Greedy Algorithms)

7.1 贪心法的基本思想

每次选择总是选出当前最优的解

例：从25,10,5,1四种硬币中选出6角1分的最少硬币数。

7.2 活动选择问题(Activity-selection Problem)

1.问题的输入

①设n个活动集合$S=\{a_1, a_2, \cdots, a_n\}$均使用同一资源且同一时间只能有一个活动占用该资源

②每个活动a_i都有一个使用该资源的开始时间s_i和结束时间f_i，且$0 \leqslant s_i < f_i < \infty$

③如果选择了活动a_i，则活动a_i的发生时间为一半开区间$[s_i, f_i)$

2. 活动a_i和a_j相容(compatible)或称不冲突

若区间$[s_i,f_i)$和$[s_j,f_j)$不重叠，则活动a_i和a_j相容，即有：

$s_i \geq f_j$ or $s_j \geq f_i$

3. 活动选择问题

存在集合$A \subseteq S$，使得$|A|$最大且A中活动互不冲突。

例：子集$A=\{a_1,a_4,a_8,a_{11}\}$以及$A=\{a_1,a_4,a_9,a_{11}\}$为最大相容子集，$|A|=4$。

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

4. 动态规划方法

step1: 描述最优子结构性质

①定义活动选择问题的子问题空间S_{ij}如下：

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

表示活动a_i结束，活动a_j开始前所有与a_i和a_j相容的活动。

为讨论方便作如下二个假设：

1）引入二个虚拟活动a_0:$f_0=0$,a_{n+1}:$s_{n+1}=\infty$；

2）所有活动按完成时间的递增序排列，即：

$f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$

由这二个假设可得出结论：当$i \ge j$时，$S_{ij}=\Phi$。

proof: 反正，假设$S_{ij} \neq \Phi$

∵ $i \ge j$时，根据假设2）有$f_i \ge f_j$

又∵$S_{ij} \neq \Phi$

∴$a_k \in S_{ij}$，按S_{ij}的定义则有：$f_i \leq s_k < f_k \leq s_j < f_j$

∴$f_i < f_j$，与假设矛盾，因此$S_{ij}=\Phi$。

这样子问题空间可进一步缩小为$0 \leq i < j \leq n+1$

②分解问题

设$S_{ij} \neq \Phi$，$a_k \in S_{ij}$

则有：$f_i \leq s_k < f_k \leq s_j$

若在S_{ij}的解中选择了a_k，则S_{ij}分解为与a_k相容的二个子问题S_{ik}和S_{kj}，即：

S_{ij}的解=S_{ik}的解$\cup S_{kj}$的解$\cup \{a_k\}$

最优自结构性质：令A_{ij}是包含a_k的S_{ij}的一个最优解，那么子问题S_{ik}和S_{kj}的解A_{ik}和A_{kj}也一定是最优的。

proof: 假设S_{ik}的解A_{ik}不是最优的，则存在一个比A_{ik}包含更多活动的解A_{ik}’，

∵ $|A_{ik}'|>|A_{ik}|$

∴ 当用A_{ik}’去替换A_{ik}则产生一个比原S_{ij}的解A_{ij}包含更多活动的解，这与A_{ij}是S_{ij}的一个最优解矛盾。

同理可证A_{kj}也是子问题S_{kj}的一个最优解。由此得到最优子结构为：

$$A_{ij}=A_{ik} \cup A_{kj} \cup \{a_k\}$$

原问题就是求$S_{0,n+1}$的最优解$A_{0,n+1}$

step2: 递归定义最优解值

令C[i,j]存放S_{ij}的最优解值$|A_{ij}|$，即$|A_{ij}|$最大

则当$S_{ij}=\Phi$ 或i≥j时，C[i,j]=0

根据最优子结构性质可以得到C[i,j]的递归式为：

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

step3 :自底向上计算最优解值

解此问题的动态规划算法与矩阵链乘问题类似。

算法时间：

∵子问题规模为$O(n^2)$，选择数为$O(n)$

∴动态规划算法时间为$O(n^3)$

5. 贪心法解此问题

定理16.1（P224）

设S_{ij}是任一非空子问题，a_m是S_{ij}中具有最早完成时间的活动，即$f_m=\min\{f_k:a_k \in S_{ij}\}$，则：

①活动a_m必定被包含在S_{ij}的某个最优解中

②子问题S_{im}是空集，使得S_{mj}是唯一可能的非空集

定理16.1的简化作用

①分解S_{ij}：二个子问题简化为一个子问题S_{mj}

②j-i-1种选择简化为一种选择：选S_{ij}中具有最早完成时间的活动

由定理16.1的简化作用使得求解问题可以采用自顶向下的方法设计算法:

①从原问题$S_{0,n+1}$中选最早完成时间的活动$a_{m1}=a_1$

②从$S_{m1,n+1}$中选最早完成时间的活动a_{m2}

③……

Recursive-Activity-Selector(s,f,i,j)

m←i+1

while m<j and s_m<f_i //在S_{ij}中找第一个与

do m←m+1 //a_i相容的活动

if m<j //$s_m \geq f_i$，找到相容的活动a_m

then return {am}∪Recursive-Activity-Selector(s,f,m,j)

else return Φ

算法执行过程如图16.1。

算法分析：用O(nlgn)的排序算法使f按递增序排列

∵算法依次从n+2个活动中挑选一个活动

∴递归调用+while循环的次数=O(n)

∴算法总时间为O(nlgn)

7.3 贪心法应用注意事项

1.应用贪心法的几项工作

①确定问题的最优子结构性质

②将优化问题转化为作出一种选择，即贪心选择

③贪心选择后应只留下一个子问题，其它子问题均为空

④证明贪心选择的正确性，即本次贪心选择与剩余子问题的最优解可以构成原问题的最优解

例：从3种硬币11，5，1中选出15分的最少硬币数。

2. 贪心选择性质

贪心选择性质是应用贪心法求解的一个必要条件。所谓选择性质是指所求问题的最优解可以通过一系列局部最优的贪心选择而得到。即：局部最优＝>全局最优

贪心选择性质是区别与动态规划方法的一个重要特征，因为贪心法在作出选择时并不依赖于将来的情况，只需根据当前的情况即可作出选择。

3. 最优子结构性质

最优子结构性质也是应用贪心法求解的一个必要条件。局部最优+一系列贪心选择产生原问题的一个最优解。

4. 贪心法与动态规划方法比较

①贪心法效率更高，表现在子问题不需要都计算，而选择数为1。

②动态规划方法的应用面更广，问题只需具有最优子结构即可，而贪心法不仅需要具有最优子结构性质，还需要具有贪心选择性质。

例如：背包问题

给定n个物品和一个背包，物品i的重量为w_i，其价值为v_i，背包可装载的总重量为W。问题是如何选择装入背包的物品，使得物品的总价值最大？

0/1背包(0-1 knapsack):物品不能拆分

零头背包(fractional knapsack)：物品可以拆分

见书P230 图16.2

7.4 Huffman编码

例如:一个包含100,000个字符的数据文件进行压缩存储，编码方式为：

	a	b	c	d	e	f
频度(千次)	45	13	12	16	9	5
等长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

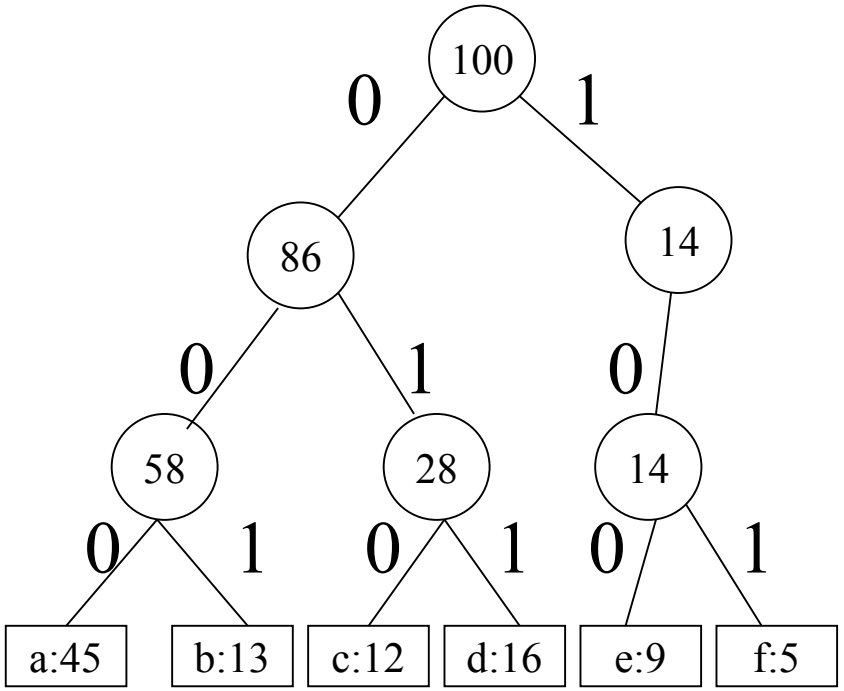
编码后文件的二进位总长为：

等长码：100,000×3＝300,000位

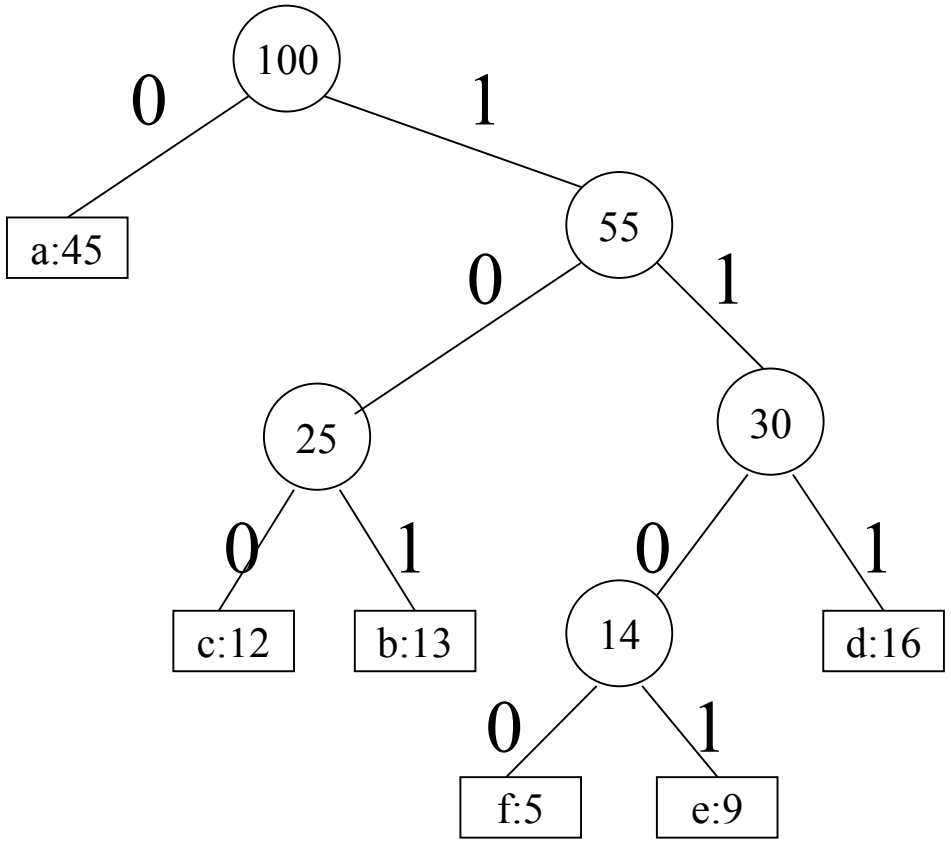
变长码：(45+(13+12+16)×3+(9+5)×4)×1000=224,000位

1. 前缀码(prefix code)

前缀码是指前缀码中任一字符编码都不是其它字符编码的前缀。



等长码



变长码

2. 最优前缀码

令C为译码字符集，T为相关前缀码树

f(c)={c∈C：字符c的频度}

$d_t(c)$：字符c在树T中的深度

$B(T)=\sum f(c).\ d_t(c)$， $c\in C$

B(T)表示所有字符编码的总长，则使B(T)值达到最小的前缀码称为最优前缀码。Huffman编码就是一种最优前缀码。

Huffman树有二个特点：

①树中所有叶结点均为译码字符，没有空余叶结点。这种树有称为满(full)二叉树

②if 叶结点数为|C| then 内部结点数为|C|−1个

3. 构造Huffman编码

```
Huffman(C)
    n←|c|
    Q←C          //Q为一队列，存放字符频度值f(c)
    for i←1 to n-1
    do  new(z)
        left[z]←x←extract-min(Q)
        right[z]←y←extract-min(Q)
        f[z]←f[x]+f[y]
        insert(Q,z)
    return extract-min(Q)
```

贪心选择策略：每次挑选二个频度值最小的字符。

例：构造包含6个字符的Huffman编码

C={f:5, e:9, c:12, b:13, d:16, a:45}

算法分析：

初始化建堆时间为：O(n)

∵当采用二叉堆管理队列时，一次extractg-min(Q)的时间为O(lgn)

∴n次抽取最小值操作时间的时间为O(nlgn)

∵一次insert(Q,z)的时间为O(lgn)

∴n次插入操作的时间为O(nlgn)

∴总时间为O(nlgn)

4. 最优前缀码的贪心选择性质和最优子结构性质

引理16.2 贪心选择性质

令C为一字符集，C中每个字符c∈C的频度为f[c]，x,y∈C是二个具有最小频度值的字符，则必定存在C的一种最优前缀码，使x,y的码长相同且仅最后一位不同。

proof：①首先证明最优前缀码树是一棵叶结点满二叉树

设T是相关于C的最优前缀码树，但T的叶结点不满

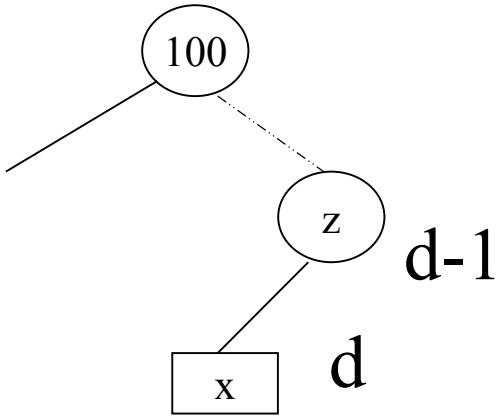
∵T是最优前缀码树

∴B(T)最小

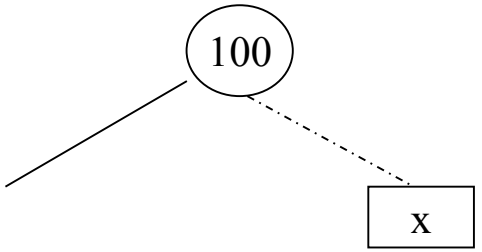
令z是T的一个内部结点且叶结点不满，则存在z的二种情况：

Z的孩子没有内部结点情况：

T：



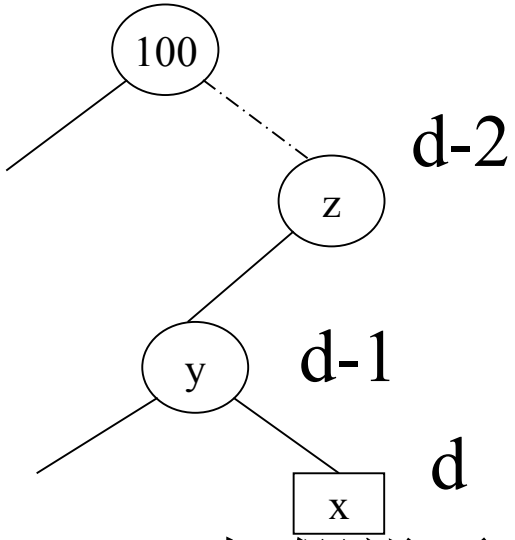
T’



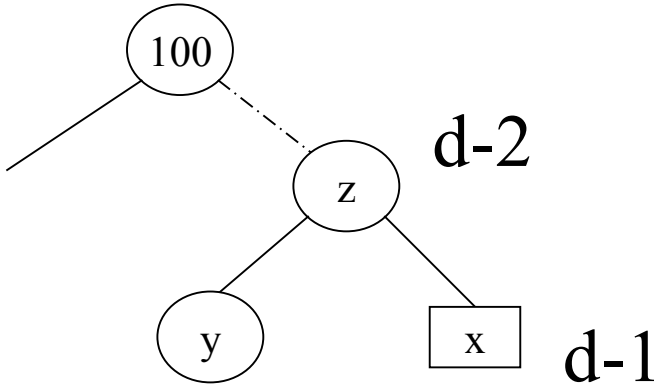
B(T’)<B(T)与假设矛盾

Z的孩子有内部结点情况：

T：



T’



B(T’)<B(T)与假设矛盾

②令T是最优前缀码树，证明x,y码长相同且仅最后一位不同

设a,b∈C是T中深度最大的一对叶结点

∵x,y是频度最小的二个字符

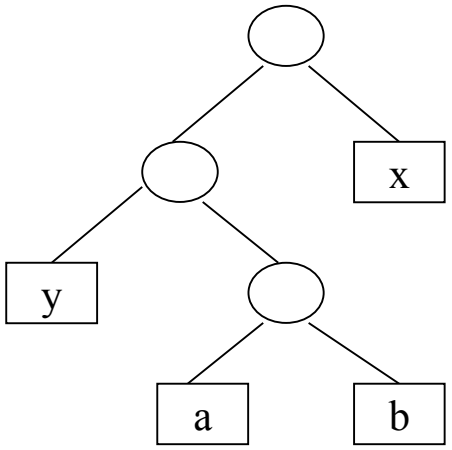
∴f[a]≥f[x]，f[b]≥f[y]

∵a,b深度最大

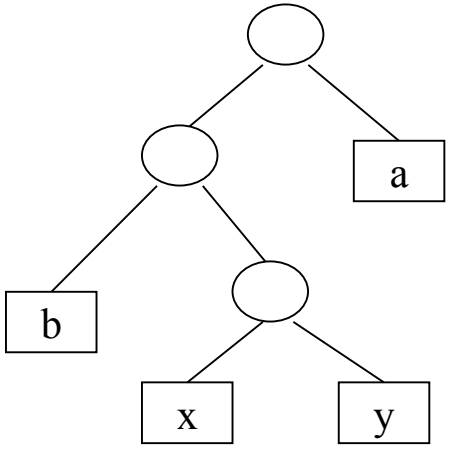
∴$d_t(a)\geq d_t(x)$，$d_t(b)\geq d_t(y)$

在树T中交换a和x，b和y的位置，得到新树T’

T



T’



∵ $d_t(a)=d_{t'}(x)$，$d_t(b)=d_{t'}(y)$，$d_t(x)=d_{t'}(a)$，$d_t(y)=d_{t'}(b)$

$$\therefore B(T)-B(T')=\sum_{c\in C} f[c]d_t(c)-\sum_{c\in C} f[c]d_{t'}(c)$$

$$=f[x]d_t(x)+f[a]d_t(a)+f[y]d_t(y)+f[b]d_t(b)$$
$$-\left(f[x]d_{t'}(x)+f[a]d_{t'}(a)+f[y]d_{t'}(y)+f[b]d_{t'}(b)\right)$$
$$=f[x]d_t(x)+f[a]d_t(a)+f[y]d_t(y)+f[b]d_t(b)$$
$$-\left(f[x]d_t(x)+f[a]d_t(a)+f[y]d_t(y)+f[b]d_t(b)\right)$$
$$=\left(f[a]-f[x]\right)\left(d_t(a)-d_t(x)\right)$$
$$+\left(f[b]-f[y]\right)\left(d_t(b)-d_t(y)\right)\geq 0$$

即 $B(T)\geq B(T')$

∵ T是一棵最优前缀码树

∴ $B(T)\leq B(T')$

联合上二式得$B(T)=B(T')$

∴ T'也是一棵最优前缀码树

∵ 最优前缀码树的叶结点是满的

∴ x,y的深度相同且仅最后一位不同

引理16.3 最优子结构性质

令C为一字符集，C中每个字符$c \in C$的频度为f[c]，$x,y \in C$是二个具有最小频度值的字符，若在C中去除字符x,y，加上频度为$f[z]=f[x]+f[y]$的新字符z，则构成一新的字符集$C'=C-\{x,y\}\cup\{z\}$。令T'是C'的一棵最优前缀码树，那么当用孩子结点为x,y的内部结点替换C'中的叶结点z后得到的新树T是一棵最优前缀码树。

proof：1）计算B(T)值

∵对所有的$c \in C-\{x,y\}$有：$d_t(c)=d_{t'}(c)$

∴$f[c]d_t(c)=f[c]d_{t'}(c)$

∵$d_t(x)=d_t(y)=d_{t'}(z)+1$

∴$f[x]d_t(x)+f[y]d_t(y)=(f[x]+f[y])d_t(x)$

$=(f[x]+f[y])(d_{t'}(z)+1)$

$=f[z]d_{t'}(z)+f[x]+f[y]$

$\because C' = C - \{x,y\} \cup \{z\}$

$$\therefore B(T) = \sum_{c \varepsilon C-\{x,y\}} f[c]d_t(c) + f[x]d_t(x) + f[y]d_t(y)$$

$$= \sum_{c \varepsilon C-\{x,y\}} f[c]d_{t'}(c) + f[z]d_{t'}(z) + f[x] + f[y]$$

$$= B(T') + f[x] + f[y]$$

或$B(T') = B(T) - f[x] - f[y]$

2)反正：假设T不是C的最优前缀码树，则

存在树T''，使$B(T'') < B(T)$

$\because$ x,y是二个频度最小的字符且$f[z] = f[x] + f[y]$

$\therefore$ 在树T''中用叶结点z替换x,y的父结点后得到树T'''

$\therefore B(T''') = B(T'') - f[x] - f[y]$

$$< B(T) - f[x] - f[y] = B(T')$$

这与T'最优矛盾，所以T是C的一棵最优前缀码树。

7.6 贪心法的理论基础

1. 胚（matroids)

def：一个胚应是满足下述条件的有序对M=(S,I)

①S是有穷非空集

②I是S的一个非空独立子集族，使得若$B \in I$且$A \subseteq B$，则$A \in I$。满足此性质的I具有遗传性，即B独立，B的子集亦独立，其中$\Phi \in I$。

③若$A \in I$，$B \in I$且$|A|<|B|$，则存在一个元素$x \in B-A$，使得$A \cup \{x\} \in I$，则称M具有交换性。

例如：由无向图$G=(V,E)$可定义$M_G=(S_G,I_G)$，其中

①$S_G=E$为G中的边集

②I_G为S_G的无回路边集族，即若$A\in E$则$A\in I_G\Leftrightarrow A$中无回路，边集A独立$\Leftrightarrow$子图$G_A=(V,A)$是一森林。

定理16.5

若G是一无向图，则$M_G=(S_G,I_G)$是一个胚。

2. 最大独立子集

给定一个胚M=(S,I)，对于I中一个独立子集A∈I，若S中有一个元素x不属于A，使得将x加入A后仍保持A的独立性，即A∪{x}∈I，则称x为A的一个可扩张元素（extension）。

当胚中的一个独立子集A没有可扩张元素时，称A是一个最大独立子集。

定理16.6 胚中所有最大独立子集大小相等

proof：设A和B是M的二个最大独立子集，且|A|<|B|

由胚的交换性可知，此时存在一个元素x∈B−A，使得A∪{x}∈I，这与A是最大独立子集矛盾。

同理可证|B|<|A|时，所以|A|=|B|。

3. 加权胚

若对M=(S,I)中的S给定一个权函数w，使得对任意的x∈S，有w(x)>0，则称M为加权胚。

S的子集A的权可定义为：

$w(A)=\sum w(x)$，$x\in A$

4. 最优子集

使w(A)权值达到最大的独立子集A称为最优子集。

例：求连通网络G=(V,E)的最小生成树问题。

定义加权胚$M_G=(S_G,I_G)$的权为w'为：

$w'(e)=W_0-w(e)>0$，其中$e\in E$，W_0为大于G中任一边权值的常数

令A是G的生成树，则$w'(A)=(|V|-1)W_0-w(A)$

若w'(A)最大则w(A)最小，所以A是G的MST。

5. 加权胚的通用贪心算法

```
Greedy(M,w)
    A←Φ
    按权值的单调减序排序S[M]   //S[M]为S的元素
    for x∈S[M]，取权值最大的w(x)
    do  if  A∪{x}∈I[M]              //独立性检测
          then A←A∪{x}
    return A
```

算法分析：

if 令n=|S|，则排序S中n个元素的时间为O(nlgn)

if 一次A∪{x}的独立性检测时间为O(f(n))

then n次独立性检测时间为O(nf(n))

∵一次抽取最大值的时间为O(lgn)

∴n次抽取最大值的时间为O(nlgn)

∴算法Greedy的时间为O(nlgn+nf(n))

引理16.7 (加权胚的贪心选择性质)

设M=(S,I)是权函数为w的加权胚，S中元素按权值单调减序排列。令x∈S是S中第一个使得{x}为独立子集的元素，则存在S的一个最优子集A，使得x∈A。

proof：①if 不存在x∈S使得{x}是独立子集

then 空集是唯一的独立子集

∴ 定理得证

②设B是S的一个非空最优子集

∵B∈I且I具有遗传性

∴B中所有单元素y所组成的子集{y}均为独立子集

又∵x是S中第一个单元素构成的独立子集且权值最大

∴对任意的y∈B，均有w(x)≥w(y)

if x∈B then 令A=B，则定理得证

else 构造包含元素x的子集A

一开始A={x}，此时A是一个独立子集

if |B|=|A|＝1 then 定理得证

else |B|>|A|，反复利用M的交换性，从B中选一个新元素加入到A中并保持A的独立性，直到|A|=|B|

∵必存在一个元素y∈B−A，使得A=B−{y}∪{x}

∴w(A)=w(B)−w(y)+w(x)≥w(B)

∵B是最优子集

∴w(B)≥w(A)

联合上二不等式可得：w(B)=w(A)

∴A也是S的一个最优子集且包含元素x。

引理16.8

设M=(S,I)是任意的胚，如果x∈S是S中某些独立子集A的一个可扩张元素，则x也是空集的一个可扩张元素。

proof：∵x是A的一个可扩张元素

∴A∪{x}∈I是独立的

又∵I具有遗传性

∴{x}独立

∴x是空集的一个可扩张元素，即{x}∪Φ={x}

推论16.9

设M=(S,I)是任意的胚，如果x∈S不是空集的一个可扩张元素，则x也不是S中任何独立子集A的可扩张元素。

proof：假设x∈S不是Φ的可扩张元素，但是独立子集A的可扩张元素

根据引理16.8可知，x应是Φ的可扩张元素与假设矛盾。

引理16.10(胚的最优子结构性质)

令x是从加权胚M=(S,I)的S中选出的第一个元素，那么原问题可简化为求加权胚M’=(S’,I’)的最优子集问题。

其中：S’={y∈S且{x,y}∈I}

I’={B⊆S−{x}且B∪{x}∈I}

M’的权函数是M的权函数在S’上的收缩。

proof：if A是M的包含x的最优子集

then A’=A−{x}∈I’是M’的一个独立子集

反之M’的一个独立子集A’产生M的一个独立子集，

即：A=A’∪{x}∈I

∴w(A)=w(A’)+w(x)

∵要使w(A)最大，则要求w(A’)最大

∴包含x的A是M的最优子集，则子问题A’也是M’的最优子集。

定理16.11（加权胚贪心算法的正确性）

令M=(S,I)是权函数为w的加权胚，则算法Greedy返回M的一个最优子集。

proof：由引理16.7可知，若算法Greedy第一次选择元素x加入A后，则必存在M的一个最优子集包含x

又∵由推论16.8可知，选择x时Greedy所抛弃的元素不可能是任一最优子集的可扩张元素

∴Greedy算法的第一次选择是正确的

由引理16.10可知，Greedy选择了x后，原问题简化为求加权胚M'的最优子集问题

∵对于M'中任一独立子集$B \in I'$均有$B \cup \{x\}$在M中独立

∴Greedy选择了x后，其后续步骤就是求加权胚M'的最优子集问题，由归纳法可知，Greedy最终求出的A是M的一个最优子集。

6. 一个任务调度问题(胚的应用)

①问题的描述

在一个单处理机上调度完成一系列任务，每个任务均可在单位时间内完成。问题的输入为：

1)集合$S=\{a_1,a_2,\ldots,a_n\}$为n个单位时间任务

2)截止期(deadline):对于$a_i \in S$,截止期为d_i且满足$1 \le d_i \le n$

3)权(处罚):对于$a_i \in S$，对应的权$w_i \ge 0$，若a_i未在d_i或d_i之前完成调度，则受处罚w_i，反之没有处罚

现要求安排一种最优调度使得总处罚最小。

②定义该问题的胚

即要定义胚M=(S,I)中S和I

S容易得到，即为输入任务集。

考虑这样一种调度：早任务优先调度

早任务：在给定的调度中能按时完成的任务

迟任务：在给定的调度中不能按时完成的任务

早任务优先调度：将早任务安排在迟任务前的调度

调度的规范形式(Canomical form)：早任务先于迟任务且早任务按截止期d_i的单调增序调度

任何一种调度都存在调度的规范形式。

例：	早任务	迟任务
原调度	a_i，……，a_j	
完成时间	k，……，k+s	
截止期	d_i，……，d_j	

其中：$s\geq 1$，k为完成的时间点

$\therefore k\leq d_i$，$k+s\leq d_j$

if $d_i>d_j$ then 交换a_i和a_j的位置

i) 交换后，a_j仍为早任务

ii) 对于a_i，∵$d_j \geq k+s$，$d_i>d_j$

∴$d_i>k+s$，即a_i仍为早任务

∵原问题是要求一种最优调度，使得总处罚最小

又∵只有迟任务才处罚

∴最优调度就是使得迟任务的总处罚最小

即：迟任务总处罚最小<===>早任务总处罚最大

∴独立集I可定义为早任务集族

def：若存在一个调度使得任务集A中没有迟任务，则$A \in I$。

另外定义一个函数$N_t(A)$表示A中deadline小于或等于t的任务个数，对于t=0,1,2,…,n时。

其中$N_0(A)=0$，$N_n(A)=|A|$，$N_t(A) \leq |A|$，对于$0 \leq t \leq n$时

引理16.12

对任意的任务集A，下列命题等价

①集合A独立

②对t=0, 1, 2, …, n都有$N_t(A) \leqslant t$

③若对A中的任务按截止期deadline单调增序调度，则A中无迟任务

Proof: ①➔②∵A独立，说明A中无迟任务，若存在某个t，使得$N_t(A)>t$，则A中必有任务要在t后完成

∴与A独立矛盾

②➔③∵A独立，说明A是某调度的早任务集，它的规范形式不会改变A的早任务性质

∴A中无迟任务

③➔①∵A中无迟任务

∴A独立

定理16.13

若S是一个具有deadline的单位时间任务集，I是所有独立子集构成的集合，则(S,I)是一个胚。

proof：

①显然S是一个有穷非空集

②遗传性：∵早任务集的任何子集仍是早任务集

∴I具有遗传性

③交换性：假设$A \in I$，$B \in I$ 且 $|A|<|B|$

令k是满足$N_t(B) \leq N_t(A)$的t的最大值

∵$N_n(B)=|B|$，$N_n(A)=|A|$，$|A|<|B|$

∴$k<n$

∴对所有的j，当$k+1 \leq j \leq n$时，$N_j(A)<N_j(B)$

即在deadline为k+1时，$N_{k+1}(B)$比$N_{k+1}(A)$包含更多的任务

令$a_i \in B-A$，deadline为k+1时的任务

令$A'=A \cup \{a_i\}$，若$A' \in I$，则问题得证

根据引理16.12②

∵A独立

∴当$0 \leq t \leq k$时，$N_t(A')=N_t(A) \leq t$

∵B独立

∴当$k+1 \le t \le n$时，$N_t(A') = N_t(A) + 1 \le N_t(B) \le t$

∴当$0 \le t \le n$时，都有$N_t(A') \le t$

∴$A' \in I$，满足交换性

∴(S,I)是一个胚

当用w_i作为a_i的权，则(S,I)是一个加权胚。

例如：

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

最优调度为：$A=\{a_2,a_4,a_1,a_3,a_7,a_5,a_6\}$

总处罚为：w5+w6=30+20=50

算法时间：∵独立性检测f(n)=O(n)

∴算法总时间为$O(n \times f(n))=O(n^2)$

第8章 平摊分析(Amortized Analysis)

8.1 平摊分析方法

平摊分析是指在某种数据结构上完成一系列操作，在最坏情况下所需的平均时间。

平摊分析与传统分析方法的主要差别为：

①平摊分析时间与传统分析方法的平均情况下时间不同，它是最坏情况下的平均时间

②平摊分析不涉及概率分析

③平摊分析中时间函数T(n)，其中n指的是操作数，而不是输入的规模

8.2 合计法(Aggregate Analysis)

合计法的思想为：把n个不同或相同的操作合在一起加以分析计算得到总时间的方法。即n个操作序列在最坏情况下的总时间，记为T(n)，其中n为操作数。

由此得到在最坏情况下每个操作的平均时间为T(n)/n。

1. 栈操作的平摊分析(不同类操作)

数据结构：栈S，初始为空

操作：		实际代价
push(S,x)：	对象x进栈	O(1)
pop(S)：	从S中弹出一个元素	O(1)
multipop(S,k)：	从S中连续弹出k个元素	O(min{|S|,k})

①传统方法分析

∵三个操作最坏的操作为multipop操作，而一次multipop操作在最坏情况下的时间为O(n)，如果n个操作均为multipop操作，则总时间将达到$O(n^2)$

∴在最坏情况下n个操作的总时间$T(n)= O(n^2)$

每个操作的平摊时间为：$T(n)/n=O(n)$

②合计法分析

∵三个操作不是相互独立的，一个对象入栈后最多弹出一次

∴在非空栈S上完成的pop次数（包括multipop操作）最多为push的次数

∵初始栈S为空，且n个操作中push的次数≤n

∴pop次数≤n

∵每个push和pop操作的实际代价为O(1)

∴n次操作在最坏情况下的总时间≤2n=O(n)

每个操作的平摊时间为$T(n)/n=O(1)$

2. 二进制计数器(同类操作)

数据结构：数组A[0..k-1]，即是一k为二进制计数器，计　数器初始值为0

操作：加1

算法：Increment(A)

```
i←0
while i<length[A] and A[i]=1
do      A[i]←0
        i←i+1
if i<length[A]        then A[i]←1
```

①传统方法分析

∵最坏情况下一次操作翻转的次数将达到O(k)

∴n次操作在最坏情况下的总时间T(n)=O(nk)

每个操作的平均时间为O(k)。

②合计法分析

直观观察计数器每一位在调用过程中0-1翻转的情况：

调用次数	A[7]…A[2]	A[1]	A[0]	本次成本	总成本
1	0	0	0	1	1
2	0	0	1	0	2	3
3	0	0	1	1	1	4
4	0	1	0	0	3	7

由上图观察可知n次操作计数器总的翻转次数≤2n

∴T(n)=O(n)

严格分析：计算计数器每一位在调用过程中的翻转次数

A[0]：每调用一次，翻转一次，共翻转n次

A[1]：每调用二次，翻转一次，共翻转 $\lfloor n/2 \rfloor$ 次

A[2]：每调用四次，翻转一次，共翻转 $\lfloor n/4 \rfloor$ 次

……

A[i]：每调用2^i次，翻转一次，共翻转 $\lfloor n/2^i \rfloor$ 次，其中 $0 \le i \le k-1$

∴n次调用计数器总的翻转次数为：

$$T(n) \le \sum_{i=0}^{k-1} n/2^i < n \sum_{i=0}^{\infty} 1/2^i = 2n = O(n)$$

∴ 每个操作的平摊时间为 $T(n)/n = O(1)$

8.3 记帐法(Accounting Method)

记帐法思想：先对每个不同的操作核定一个不同的费用（时间），然后计算n次操作总的费用。

这种事先的操作费用核定可能与实际费用不符，存在二种情况：

超额收费：核定费用>实际费用，此时差额存放在该对象的身上

收费不足：核定费用<实际费用，此时差额由该对象身上的存款支付

费用的核定(平摊核定)是否正确需检查n次操作核定总费用是否大于n次操作实际总费用

若令C_i为第i次操作的实际费用

$\hat{C}_i$ 为第i次操作的平摊费用

那么要求：

$$\sum_{i=1}^{n}\hat{C}_i \geq \sum_{i=1}^{n} C_i \quad \text{(对任意的n)}$$

或改写为：

$$\sum_{i=1}^{n}(\hat{C}_i - C_i) \geq 0 \quad \text{(对任意的n)}$$

1. 栈操作

三个不同操作的平摊核定如下：

	实际费用	核定费用
push	1元	2元
pop	1元	0元
multipop	min{|S|,k}	0元

假设1元代表O(1)的工作量

检查平摊费用的核定：

入栈(push)：核定2元，其中1元用于支付对象入栈时的实际费用，另外1元作为存款存放在该入栈对象的身上。

出栈(pop,multipop)：核定0元，对象出栈的实际费用由该对象身上的存款支付。

∵初始栈为空，入栈先于出栈，此时栈中每个对象身上均有1元存款

∴执行pop操作时出栈对象身上均有1元存款可用于支付出栈的实际费用

∴任何时刻$\sum(C^{\wedge}_i - C_i) \geq 0$

∵最坏情况下n个操作均为push操作

∴n次操作的总费用$T(n) \leq 2n = O(n)$

∴n个push、pop、multipop操作的平摊时间为O(1)。

2. 二进制计数器

平摊费用核定：

∵每个操作的实际成本为0-1翻转的位数，令翻转1次的实际费用为1元

置位操作(0→1)：核定2元，1元用于支付实际成本，另一元作为存款存放在刚完成置位的1身上

复位操作(1→0)：核定0元，1元的实际成本由其身上的存款支付

平摊费用核定验证：

∵计数器初态为0

∴置位先于复位

∵置位时2元中1元支付置位时的实际成本，另1元作为存款存放在该置位的1身上

∴任何时刻计数器每位1身上均有1元存款可支付复位时的实际成本

$\therefore \sum(C_i\hat{}-C_i)\geq 0$

∵最坏情况下n个操作均为置位操作且总置位次数≤n

∴总成本$T(n)\leq 2n=O(n)$

每个操作的平摊时间为$O(1)$。

8.4 势能法(potential method)

思想：通过定义一个势函数完成对每个操作的平摊核定

令n个操作的数据结构为D，数据结构的初态为D_0

C_i：表示第i次操作的实际成本

$\hat{C}_i$：表示第i次操作的平摊成本

D_i：表示在数据结构状态为D_{i-1}上完成第i次操作后的数据结构状态，op_i

即：　　$D_{i-1}=====>D_i$，$i=1,2,\ldots,n$

Φ：势函数

$\Phi(D_i)$：表示将Di映射为一实数，这个实数值称为势能

用势函数Φ表示第i次操作的核定费用C_i^定义如下：

$C_i^{\wedge} = C_i + \Phi(D_i) - \Phi(D_{i-1}) = C_i + \Phi_i - \Phi_{i-1}$

其中$\Phi_i - \Phi_{i-1}$称为势差，势差的不同结果类似与记帐法：

$\Phi_i - \Phi_{i-1} > 0$：超额收费

$\Phi_i - \Phi_{i-1} < 0$：收费不足

如同记帐法需对操作的平摊核定进行验证，为保证势函数的正确性，也需验证平摊总费用是否是实际总费用的上界，即：

$$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} (C_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} C_i + \Phi_n - \Phi_0$$

$$\because \text{要求} \sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$

$$\therefore \text{对任意的} n, \text{要求} \Phi_n - \Phi_0 \geq 0$$

1. 栈操作

定义势函数Φ：栈中的元素个数

验证势函数：

∵初始栈为空

∴$\Phi_0=0$

∵任何时刻栈中的元素个数≥0

∴对任意的n，$\Phi_n\geq 0$

平摊分析：令实际代价为1，表示O(1)的工作量

①push操作：

设D_{i-1}时，栈S中的元素个数为|S|，则push一个元素后D_i中的元素个数为|S|+1，即$\Phi_{i-1}=|S|$，$\Phi_i=|S|+1$

∵$C_i=1$

∴$C_i\hat{}=C_i+\Phi_i-\Phi_{i-1}=1+|S|+1-|S|=2$

②pop操作：

∵C_i=1，令Φ_{i-1}=|S|，则Φ_i=|S|−1

∴$C_i^{\wedge}$=C_i+Φ_i−Φ_{i-1}=1+|S|−1+|S|=0

③multipop操作：

令Φ_{i-1}=|S|，C_i=k’=min(|S|,k)，则Φ_i=|S|−k’

∴$C_i^{\wedge}$=k’+|S|−k’+|S|=0

∵一次操作在最坏情况下为push操作

∴n次操作在最坏情况下均为push操作的总时间为：

T(n)=2n=O(n)

每个操作的平摊时间为O(1)。

2. 二进制计数器

定义势函数Φ：计数器中1的个数

验证势函数：

∵计数器初态为0

∴$\Phi_0 = 0$

∵任何时刻计数器中1的个数≥0

∴对任意的n，$\Phi_n \geq 0$

若计数器初态不为0时，则

令初始势能$\Phi_0 = b_0 > 0$，$\Phi_n = b_n$

∵$0 \leq b_0$，$b_n \leq k$

$$\therefore \sum_{i=1}^{n} C_i = \sum_{i=1}^{n} \hat{C}_i - \Phi_n + \Phi_0$$

$$\leq \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0 \leq 2n + b_0$$

$$\leq 2n + k \leq 3n = O(n) \quad \text{当} k = O(n)$$

8.5 动态表(Dynamic tables)

表扩张(table expansion)：发生在插入一个元素x时，此时表满没有空间存放x，所以将引起表扩张。表扩张的步骤为：

①申请一块更大表

②拷贝原表到新表中

③释放原表

④在新表中插入x

表收缩（table contraction)：发生在删除一个元素x后，表中元素个数小于某个设定值时，引起表收缩。表收缩步骤为：

①删除元素x

②申请一块更小的表

③拷贝原表到新表中

④释放原表

定义一个装填因子(load factor)α(T)描述动态表当前的状态：

①非空表：$\alpha(T)=num[T]/size[T]$　　$0\leq\alpha\leq1$

num[T]：表中元素个数，size[T]：表的大小

②空表：size[T]=0，此时定义$\alpha(\varphi)=1$

注意空表指的是表的大小为0的情况

1. 表扩张

假设对动态表的操作均为一系列的插入操作，此时动态表只存在表扩张的情况。表扩张的自然策略为：$\alpha=1$时表扩张一倍。刚完成扩张时$\alpha=1/2$，其它情况下$\alpha>1/2$，所以有$\alpha\geq 1/2$。

```
Table-Insert(T,x)
    if size[T]=0 then size[T]←1      //分配大小为1的表空间
    if num[T]=size[T]                                    //表满
    then 分配大小为2×size [T]的新表new-table
            copy table[T] to new-table中
            free table[T]
            table[T]←new-table
            size[T]← 2×size [T]
    x插入到table[T]中
    num[T]←num[T]+1
```

算法分析：假设插入一个元素到表中的实际代价为1，初始表$T_0=\varphi$

①传统方法分析

②合计法分析

③记帐法

④势能法

定义势函数Φ：$\Phi(T)=2num[T]-size[T]$

2. 表扩张和收缩

对动态表既有插入又有删除操作时，α值设定的理想目标如下：

①α有一个常数下界

②操作的平摊代价有一个常数上界

为了达到这个理想目标，需要制定动态表合适的扩张与收缩策略，首先检查最自然的策略。

1）自然策略

表满时插入一个元素($\alpha=1$)，此时表扩张一倍

表半满时删除一个元素($\alpha=1/2$)，此时表收缩一半。

2）改进后的策略

$\alpha=1$时，插入一个元素表扩张一倍

$\alpha=1/4$时，删除一个元素表收缩一半

定义势函数

$$\Phi(T)=\begin{cases}2num[T]-size[T] & \text{if } \alpha\geq 1/2\\ size[T]/2-num[T] & \text{if } 1/4\leq\alpha<1/2\end{cases}$$

第九章 二项堆(Binomial Heaps)

二项堆、Fib堆以及二叉堆均可归类为可归并堆(mergeable heap)，这类堆的特点是能支持5个基本操作和二个扩展操作。

①make-heap()：创建空堆

②insert(H,x)：结点x插入到堆H中

③minimum(H)：取最小值

④extract-min(H)：抽取最小值

⑤union(H_1,H_2)：合并堆H_1，H_2

⑥decrease-key(H,x,k)：x减值为k

⑦delete(H,x)：从堆H中删除结点x

三个堆实现以上三个7个操作的时间如图19.1，P277

9.1 二项树(Bin0mial trees)

1. 二项树定义

def：仅包含一个结点的有序树是一棵二项树称为B_0树。二项树B_k由二棵B_{k-1}树组成，其中一棵B_{k-1}树的根作为另一棵B_{k-1}树根的最左孩子($k\geq 0$)。

2. 二项树性质

引理19.1 二项树B_k具有性质：

①有2^k个结点 $n=2^k$

②树的高度为k $k=\lg n$

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

④根的度最大且为k，若根的孩子从左到右编号为k-1,k-2,…,1,0，则孩子i恰好是子树B_i的根。

推论19.2

在一棵具有n个结点的二项树中,任一结点的最大度为lgn。

9.2 二项堆(binomial Heaps)

1. def：二项堆H是一个满足下述条件的二项树的集合：

①H中的每棵二项树满足最小堆性质

②对任意的非负整数k，H中至多有一棵二项树根的度为k

2. 二项树的逻辑表示

例：H中有13个结点，即n=13，则$B_H=1101_2$

head[H]------->B_0-------->B_2-------->B_3

二项堆中的所有二项树由一根表链接，根表的头指针用head[H]表示。

①根表为一单链表

②根表链接所有二项树的根结点

③根表按照度的递增序链接

3. 结点形式

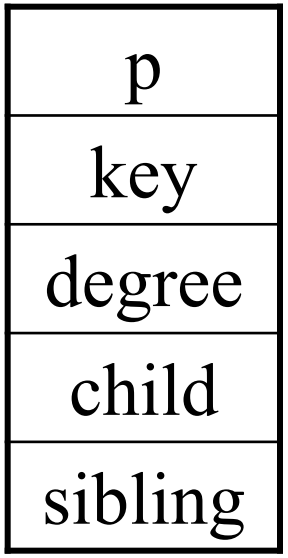
每个结点包含5个域：

指针p指向父结点

degree：孩子个数

child：指向最左孩子

sibling：指向右兄弟



见书P280,图19-3

9.3 二项堆操作

1. 创建空堆操作

```
Make-Binomial-Heap()
    head[H]←nil
    return H
```

2. 取最小值操作

```
Binomial-Heap-Minimum(H)
    y←nil，x←head[H]，min←∞
    while x≠nil
    do  if  key[x]<min
        then min←key
                y←x
        x←sibling[x]
    return y
```

3. 堆合并操作

Binomial-Heap-Union(H_1,H_2)

该操作把二个堆H_1，H_2合并为一个新堆，合并过程分为二步：

①把二个有序根表合并为一个新的有序根表，如图19.5所示。二个根表合并的操作算法为Binomial-Heap-Merge(H_1,H_2)。

②消除相同度的根

消除方法：y,z分别指向二棵不同的B_{k-1}树，消除过程就是合并这二棵树使之成为一棵B_k树。消除算法为：Binomial-Link(y,z)。

合并实例如图19.5

4. 插入一个结点操作

Binomial-Heap-Insert(H,x)

5. 抽取最小值操作

Binomial-Heap-Extract-Min(H)

实例见图19.7

6. 减值操作

Binomial-Heap-Decrease-Key(H,x,k)

7. 删任意结点操作

Binomial-Heap-Delete(H,x)

第十章 Fibanocci堆

10.1 Fib堆

1. def：Fib堆是一具有最小堆有序的树的集合。

Fib堆和二项堆比较：

相似点：

①均支持5个基本操作和2个扩展操作

②均为树的集合

③堆中每棵树均满足最小堆性质

④均采用根表链接堆中所有树根结点

不同点：

①Fib堆根表中度不再唯一

②Fib堆中树不要求一定是二项树

③Fib堆根表不需要按照度的递增序排列

④Fib堆根表头指针指向根表中具有最小值的树根结点

⑤Fib堆操作的时间分析采用平摊分析方法

2. Fib的逻辑表示

见书P292,图20-1

Fib堆根表的特点：

①根表头指针改用Min[H]表示，且总是指向堆中具有最小值的结点

②根表及每棵树的兄弟间均采用双向循环链表形式

3. 结点形式

Fib堆的每个结点包含7个基本域：

p	left	right	key	degree	child	mark

其中：

p：指向其父结点指针

left：指向左兄弟指针

right：指向右兄弟指针

child：指向其任一孩子结点指针

mark[x]：为一布尔量，表示结点x自从成为另一结点的孩子后，x是否失去了一个孩子。当创建新结点以及x成为另一结点的孩子时，mark[x]置为false，当x失去一个孩子时，mark[x]置为true.

4. 势函数

所有Fib堆操作均采用统一的势函数定义如下：

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

其中：t(H)表示堆中树的个数

m[H]表示mark域标记为true的结点数

5. 最大度

设Fib堆中人一结点的最大度的上界为D(n)

对操作的要求为：5个基本操作$D(n)\leq \lg n$

2个扩展操作$D(n)=O(\lg n)$

6. 无序二项树(unordered binomial tree)

def：U_0树只包含一个结点，U_k树是由二棵无序U_{k-1}树组成，其中一个无序U_{k-1}树的根作为另一棵无序U_{k-1}树的孩子。

10.2 Fib堆的操作

1. 创建空堆操作

Make-Fib-Heap()

min[H]←Nil

n[H]←0

return H

2. 插入一个结点的操作

Fib-Heap-Insert(H,x)

在Fib堆中插入一个结点的步骤为：

①x作为新树插入根表中

②检查Min[H]头指针

③新结点x的mark域置为false

3. 取最小值操作

Fib-Heap-Minimum(H)

4. 合并堆操作

Fib-Heap-Union(H_1,H_2)

5. 抽取最小值操作

Fib-Heap-Extract-Min(H)

实现步骤：

step1：将被删结点z的所有孩子作为新树插入到根表中

step2：将z从堆H中删除

Step3：调整根表

```
Consolidate(H)
    for i←0 to D(n[H])              //A[0..D(n)]为一指针数组
    do  A[i]←nil                    // A[i]指向度为i的根结点
    for 根表中的每个结点w           //依次扫描根表
    do  x←w
        d←degree[x]
        while A[d]≠nil              //此时A[d]中已有和x度相同的结点
        do   y←A[d]
             if key[x]>key[y] then exchange(x,y)
             Fib-Heap-Link(H,y,x)
             A[d]←nil
             d←d+1
        A[d]←x
```

6. 减值操作

Fib-Heap-Decrese-key(H,x,k)

Fib堆在O(1)时间界内完成减值操作的具体步骤：

①if x为根结点　then 减值不违背最小堆性质，只需检查min[H]指向即可

②if x为非根结点　then 令y=p[x]

此时若key[x]<key[y]，则将x为根的子树从y上删除并插入到根表中

存在问题？

改进的方法：

前二步相同，当x为根的子树从y上删除并插入到根表中后，if y为非根结点且删x是删除y的第二个孩子

then 将y子树从其父结点z=p[y]上删除，if 删y是删z的第二个孩子是时，then z子树也将从其父结点上删除。重复此过程，直到被删结点的父结点为根结点或仅是失去第一个孩子为止。这种减值过程是一种连续删除过程，由算法Cascading-Cut完成。

完整算法见书P301，及图例20-4

```
Cascading-Cut(H,y)
   z←p[y]
   if   z≠nil
   then if  mark[y]=false          //y尚未失去过孩子
          then  mark[y]←true       //y失去了第一个孩子x
          else   Cut(H,y,z)        //y已失去了二个孩子
                 Cascading-Cut(H,z)   //重复向上处理
```

注意算法中mark域的变化，同时需关注算法Cut中mark域的变化。

7. 删任一结点操作

Fib-Heap-Delete(H,x)

Fib-Heap-Decrese-key(H,x,-∞)

Fib-Heap-Extract-Min(H)

平摊时间：O(D(n))

10.3 最大度的界

本节将证明任一结点的最大度D(n)=O(lgn)

令size(x)为以x为根的子树的所有结点数（包括x本身）

引理20.1

令x为Fib堆中的任一结点，并设degree[x]=k。令x的孩子$y_1,y_2,\ldots,y_k$是按连接到x的时间先后排列，则有degree[y_1]≥0且degree[y_i]≥i−2，当i=2,3,…,k

引理20.2　对所有的$k \geq 0$，有$F_{k+2} = \sum_{i=0}^{k} F_i$，其中

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

引理20.3

令x为Fib堆中任一结点，k=degree[x]，则size(x)≥F_{k+2}≥Φ^k，其中$\Phi=(1+\sqrt{5})/2$。

推论20.4

在具有n个结点的Fib堆中，任一结点的最大度D(n)=O(lgn)。

第11章 最大流(Maximum Flow)

11.1 流网络(flow network)

1. 流及流网络

1) 流网络G=(V,E)为一有向图，图中每条边(u, v)∈E均为非负容量c(u,v)≥0。如果(u,v)不属于E，则c(u,v)=0。

在流网络中有两个特殊的顶点称为源(source)和汇(sink),分别用s和t表示。

2) 流：在图G中，流是一实函数f：V×V→R，并满足以下三个条件：

容量约束(capacity constraint)：

对于所有的u, v∈V，f(u,v)≤c(u,v)

反号对称(skew symmetry):

对于所有的u, v∈V，f(u,v)= −f(v,u)

流守恒(flow conservation)：

对所有的$u \in V-\{s,t\}$，$\sum_{v\in V} f(u,v)=0$

3)流函数$f(u,v)$：表示从顶点u到顶点v的流量值，可以为正,0及负。

特别地，网络的流量值记为：

$|f|=\sum_{v\in V} f(s,v)$，即流出源s的流量总和。

见书P397，图26.1

4) 根据流的反号对称性质，流守恒又可表示为流进一个顶点的流量总和为0，即：对所有的$v \in V-\{s,t\}$，$\sum_{u\in V} f(u,v)=0$

5) 如果$(u,v)\notin E$，$(v,u)\notin E$，则$f(u,v)=f(v,u)=0$

6)流入顶点v的总正流：

对所有的$v \in V-\{s,t\}$，$\sum_{\substack{u\in V \\ f(u,v)>0}} f(u,v)$

某个顶点的总净流定义为：

净流＝流出总正流－流入总正流

除了源和汇外，顶点的净流值等于0，所以流守恒可表述为“流进＝流出”。

2. 多源多汇网络

对多源多汇网络通过增加二个特殊顶点，超级源s和超级汇t，并分别定义从超级源s到其它源s_i及其它汇t_i到超级汇t的容量为$c(s,s_i)=\infty$，$c(t_i,t)=\infty$，则多源多汇问题就转换为单源单汇问题。

3. 流函数的扩展表示

如：$f(X,Y)=\sum_{x\in X}\sum_{y\in Y}f(x,y)$

流守恒：对所有的$v\in V-\{s,t\}$,有$f(u,V)=0$

引理26.1

令$G=(V,E)$为一流网络，f是图G的一个流，则有

1）对所有的$X\subseteq V$，有$f(X,X)=0$

2) 对所有的$X,Y\subseteq V$，有$f(X,Y)=-f(Y,X)$

3) 对所有的$X,Y,Z\subseteq V$且$X\cap Y=\Phi$，有$f(X\cup Y,Z)$

$=f(X,Z)+f(Y,Z)$及$f(Z,X\cup Y)=f(Z,X)+f(Z,Y)$

例：证明$|f|=f(V,t)$

- 11.2 Ford-fulkerson方法
- Ford-Fulkerson-Method(G,s,t)
- 初始化流f=0
- while 存在增广路径P
- do 沿增广路径P增加流f
- return f

- Ford-fulkerson方法需解决以下几个问题：
- ①二个流相加是否仍是一个流及如何构造流？
- ②如何沿增广路径增加流？
- ③在流网络中没有了增广路径是否得到了最大流？

- 1. 剩余网络(residual network)
- 所谓剩余网络就是在流网络中除去某个流后，所有剩余容量(residual capacity)组成的网络。
- 对于二个顶点u,v间的剩余容量定义为：
- $c_f(u,v)=c(u,v)-f(u,v)$

- 则剩余网络可定义为：
- 对于给定的流网络G=(V,E)以及流f，则$G_f(V,E_f)$是相对与流f的剩余网络。其中
  $E_f=\{(u,v)\in V\times V：c_f(u,v)>0\}$。

- E_f的构成：
- ① if f(u,v)<c(u,v)
- ② if f(u,v)>0
- 由流网络到剩余网络的构造见P401，图26-3
- 原流网络中边数与剩余网络的边数满足关系：$|E_f| \leq 2|E|$

- 引理26.2
- 令G=(V,E)是一个源为s，汇为t的流网络，f是G中的一个流。令G_f是相关于f的剩余网络，f’是G_f的一个流，则f+f’也是G中的一个流且流值$|f+f'|=|f|+|f'|$。

- 2. 增广路径(Augmenting Path)
- 所谓增广路径是指在G_f中一条从s到t的简单路径。根据剩余网络的定义，在增广路径中的每条边在不违背容量约束的条件下可以增加一定量的正流。如图26.3(b)，P401

- 在一条增广路径P中可以增加流的最大量称为增广路径P的剩余容量，定义为：
- $c_f(P)=\min\{c_f(u,v):(u,v)\in P\}$

- 引理26.3
- 令f是流网络G=(V,E)的一个流，令P是剩余网络G_f的一条增广路径，定义函数f_p：$V\times V\rightarrow R$，并有：
- $$f_p(u,v)=\begin{cases} c_f(P) & \text{if } (u,v)\in P \\ -c_f(P) & \text{if } (v,u)\in P \\ 0 & \text{otherwise} \end{cases}$$
- 那么f_p是G_f的一个流且流值为$|f_p|=c_f(P)>0$

- 推论26.4
- 令f是流网络G=(V,E)的一个流，P是G_f中的一条增广路径，f_p如上定义。定义函数f’:$V\times V\rightarrow R$且$f'=f+f_p$，那么f’是G中的一个流，其流值$|f'|=|f|+|f_p|>|f|$。

- 3. 流网络的截(cut)
- 流网络的截(S,T)是顶点V的一种划分，其中T=V−S，s∈S，t∈T。
- 相关于截(S,T)的表述：
- ①流函数f(S,T)：穿越截(S,T)的净流，即流出总正流－流入总正流。
- ②C(S,T)：截(S,T)的容量，即从S到T的容量之和。
- ③最小截：所有截中容量最小的截
- 如P403，图26.4
- 例：截S={s,v_1,v_2}，T={v_3,v_4,t}
- f(S,T)=19，c(S,T)=26
- 例：截S={s,v_1,v_2,v_4}，T={v_3,t}
- f(S,T)=19，c(S,T)=23

- 引理26.5
- 令f是源为s汇为t的流网络G中的一个流，(S,T)是G的一个截，那么穿越截(S,T)的净流f(S,T)=|f|。

- 推论26.6
- 流网络G中的任何流值均受限于G中截的容量限制。

- 定理26.7
- 如果f是源为s汇为t的流网络G中的一个流，那么以下条件相等：
- ①f是G中的最大流
- ②剩余网络没有增广路径
- ③G中存在截(S,T)，使|f|=C(S,T)

- 由最大流最小截定理可对Ford-Fulkerson方法细化，算法见书P404及图例26.5

4. Edmonds-Karp算法

Edmonds-Karp算法是Ford-Fulkerson方法的一种具体实现。它采用BFS搜索方法在剩余网络中寻找一条增广路径。具体做法是先对每条边的权值统一赋值为1，然后找一条从s到t的最短路径。基于此方法，Edmonds-Karp算法求最大流的时间为$O(VE^2)$。

5. 最大二分图匹配(Maximum Bipartite Matching)

①匹配

对于给定的无向图G=(V,E)，M是边集E的一个子集。如果M中最多有一条边关联于顶点$v \in V$，则称M为一个匹配。如果M是所有匹配中具有最大目(cardinality)的匹配，则称M为最大匹配，即对任意的匹配M’，有$|M| \geq |M'|$。

②二分图

如果把顶点集V划分为二个不相交的子集合L和R，即V=L∪R，L∩R=Φ，并且E中的所有边均分别连接L和R中的顶点，则称这样的图为二分图。即：

{(u,v)∈E：u∈L，v∈R}

二分图中具有最多边的匹配称为最大二分图匹配。

③求二分图的最大匹配问题

此问题可以用最大流算法求解。首先需把二分图转换为流网络：

1)增加一个源s和汇t及s到L中所有顶点的边以及R中所有顶点到t的边

2)把无向图改为有向图，方向为s→L→R→t

3) 给所有边赋值为单位容量值

这样二分图G=(V,E)就转换为流网络G’=(V’,E’)，其中：

V’=V∪{s,t}

E’={(s,u):u∈L}∪{(u,v):u∈L,v∈R,(u,v)∈E}∪{(v,t):v∈R}

∵|E’|=|E|+|V|≤|E|+2|E|=3|E|=Θ(E)

图例见书P410，图26.8

引理26.10

令G=(V,E)为一二分图，V=L∪R，G’=(V’,E’)是相关G的流网络。如果M是G中的一个匹配，则在G’中存在一个整数流值|f|=|M|。相反，如果在G’中存在整数流f，则在G中存在一个匹配M，使得|M|=|f|。

proof: ①令M是G的一个匹配

定义f为： $f(s,u)=f(u,v)=f(v,t)=1$ $if(u,v)\in M$

$f((u,s)=f(v,u)=f(t,v)=-1$ $if(u,v)\in M$

$f(u,v)=0$ $if(u,v)\in E'$and (u,v) not$\in M$

∵ $f(u,v)\leq c(u,v)=1$ 满足容量约束

$f(u,v)= -f(v,u)$ 满足反号对称

$f(u,V)=0$ $u\in V-\{s,t\}$ 满足流守恒

∴f是G'的一个流

又∵$f(u,v)=c(u,v)=1$ $(u\in L,v\in R$ and $(u,v)\in M)$

∴截$(L\cup\{s\},R\cup\{t\})$的净流$=|M|$

即： $f(L\cup\{s\},R\cup\{t\})=|M|$

根据引理26.5可知， $|f|= f(L\cup\{s\},R\cup\{t\})=|M|$

②if f是G’的整数流

令M={(u,v):u∈L,v∈R,f(u,v)>0}

∵ f为整数流且c(u,v)=1

∴ f(u,v)=c(u,v)=1，当(u,v)∈M

根据流守恒，除s,t外，顶点的净流值为0

即：在每条s→u→v→t路径中的顶点u∈V-{s,t}不会出现在其他路径中，因此M是一个匹配

∴ |M|=f(L,R)=f(L,V’)-f(L,L)-f(L,s)-f(L,t)

=f(L,s)=f(s,L)=f(s,V’)=|f|

推论26.12

二分图G的一个最大匹配M的目数等于其相关流网络G’的最大流f的流值。

proof：假设M是G的最大匹配，而f不是G’的最大流，则在G’中一定存在一个最大流f’，使得|f’|>|f|

根据引理26.10，在G中存在一个匹配M’，使得

|M’|=|f’|>|f|=|M| 与M为最大匹配矛盾。

算法分析：

∵ c(u,v)=1

∴ while循环最多执行O(M)=min(L,R)=O(V)次

即|f*|=O(V)

∴ 二分图最大匹配算法时间为O(VE’)=O(VE)。