第3章　动态规划

回顾：动态规划法 .VS. 分治法

- 动态规划法的实质也是将较大问题分解为较小的同类子问题，这一点上它与分治法类似。
- 分治法的子问题相互独立，相同的子问题被重复计算。
- 动态规划法利用问题的最优子结构特征，设计自底向上的计算过程，通过从子问题的最优解逐步构造出整个问题的最优解，避免重复计算。

回顾：动态规划基本步骤

- 设计一个动态规划算法，通常可以按以下几个步骤进行：

- **找出最优解的性质，并刻划其结构特征。**
- **递归地定义最优值。**
- **计算出最优值，通常采用自底向上的方式。**
- **根据计算最优值时得到的信息，构造最优解。**

回顾：动态规划基本步骤

通过应用范例学习动态规划算法设计策略。

（1）矩阵连乘问题；

（2）最长公共子序列；

（3）凸多边形最优三角剖分；

（4）多边形游戏；

3.7 图像压缩

在计算机中，常用像素点的灰度值序列$\{p_1,p_2,……p_n\}$表示图像。其中整数p_i,1<=i<=n，表示像素点i的灰度值。通常灰度值的范围是0~255。因此最多需要8位表示一个像素。

- 一张分辨率为640 x 480的图片，那它的像素就达到了307200，也就是人们常说的30万像素

图像压缩

图象的变位压缩存储格式将所给的像素点序列 $\{p_1,p_2,\ldots,p_n\}$, $0\le p_i\le 255$分割成m个连续段$S_1,S_2,\ldots,S_m$。

第i个像素段S_i中($1\le i\le m$)，有l[i]个像素,且该段中每个像素都只用b[i]位表示。

分段的过程就是要找出断点，让一段里面的像素的最大灰度值比较小，那么这一段像素(本来需要8位)就可以用较少的位(比如7位)来表示，从而减少存储空间。

设 $t[i]=\sum_{k=1}^{i-1} l[k]$ 则第i个像素段Si为

$$S_i=\{p_{t[i]+1},\cdots,p_{t[i]+l[i]}\}$$

比如某个片段为：

$p_i=10$, $p_{i+1}=15$, $p_{i+2}=100$, $p_{i+3}=55$, $p_{i+4}=200$, $p_{i+5}=255$

可以分成p_i-p_{i+3}和p_{i+4}-p_{i+5}两段，而第一段的每个像素只需要7个比特位就可以表示。

图像压缩

本题中， $0 \le p_i \le 255$ ， 因此$b[i] \le 8$，即需要用3位表示b[i]，如果限制每段不超过255个像素，即$1 \le l[i] \le 255$，则需要用8位表示l[i]。因此，第i个像素段所需的存储空间为l[i]*b[i]+8+3= l[i]*b[i]+11位。

按此格式存储像素序列$\{p_1,p_2,\ldots,p_n\}$，需要 $\sum_{i=1}^{m} l[i]*b[i]+11m$ 位的存储空间。

图象压缩问题要求确定像素序列$\{p_1,p_2,\ldots,p_n\}$的最优分段，使得依此分段所需的存储空间最少。每个分段的像素不超过256个。

图像压缩

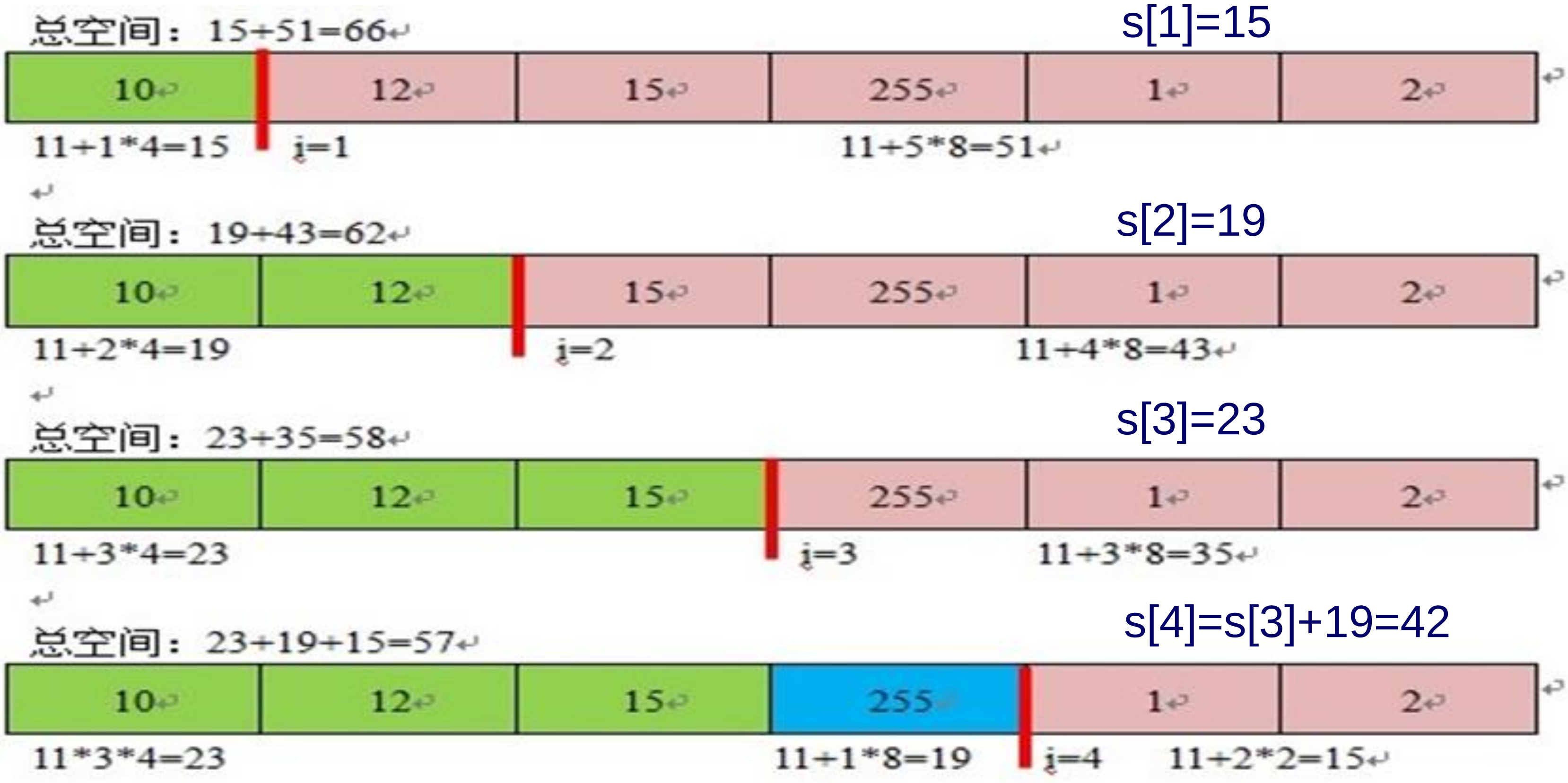
设$l[i]$，b[i]， $1\leq i\leq m$ ，是$\{p_1,p_2,\ldots,p_n\}$的最优分段。显而易见，$l[1]$，b[1]是$\{p_1,\ldots,p_{l[1]}\}$的最优分段，且$l[i]$，b[i]， $2\leq i\leq m$是$\{p_{l[1]+1},\ldots,p_n\}$的最优分段。即图象压缩问题满足最优子结构性质。

设s[i]，$1\leq i\leq n$，是象素序列$\{p_1,\ldots,p_i\}$的最优分段所需的存储位数，**则s[i]为前i-k个的存储位数（已算过，是s[i-k]）加上后k个的存储位数（需再计算）。**由最优子结构性质易知：

$$s[i]=\min_{1\leq k\leq \min\{i,256\}}\{s[i-k]+k*b\max(i-k+1,i)\}+11$$

其中 $b\max(i,j)=\left\lceil \log\left(\max_{i\leq k\leq j}\{p_k\}+1\right)\right\rceil$ 为p_i到p_j中，最大的值需要的比特位数。

例：6个像素，值分别为10,12,15,255,1,2



算法复杂度分析：
由于算法compress中对k的循环次数不超过256，故对每一个确定的i，可在时间O(1)内完成的计算。因此整个算法所需的计算时间为O(n)。

```
void Compress(int n, int p[], int s[], int l[], int b[])  //令s[i]为前i个段最优合并的存储位数
{
    int Lmax = 256, header = 11;
    s[0] = 0;
    for(int i=1; i<=n; i++)  //i表示前几段
    {
        b[i] = length(p[i]); //计算像素点p需要的存储位数
        int bmax = b[i];
        s[i] = s[i-1] + bmax;  //故下面j从2开始
        l[i] = 1;
        for(int j=2; j<=i && j<=Lmax; j++)
        //递推关系:s[i]=  min {s[i-j]+ lsum(i-j+1,i)*bmax(i-j+1,i) } + 11
        {
            if(bmax < b[i-j+1])
                bmax = b[i-j+1];
            if(s[i] > s[i-j] + j*bmax)
            //因为一开始所有序列并没有分段,所以可以看作每一段就是一个数,故lsum(i-j+1, i) = j;
            {
                s[i] = s[i-j] + j*bmax;
                l[i] = j;
                 //最优断开位置,l[i]表示前i段的最优分段方案中应该是在i-j处断开
                 //比如l[5] = 3,这表示前五段的最优分段应该是(5-3=2)处断开,s[5] = s[2] + 3*bmax
                 //即 12 | 345,以此类推,得到l[n];之后构造最优解时再由l[n]向前回溯
            }
        }
        s[i] += header;
    }
}
```

3.9 流水作业调度

n个作业{1，2，…，n}要在由2台机器M_1和M_2组成的流水线上完成加工。每个作业加工的顺序都是先在M_1上加工，然后在M_2上加工。M_1和M_2加工作业i所需的时间分别为a_i和b_i。

流水作业调度问题要求确定这n个作业的最优加工顺序，使得从第一个作业在机器M_1上开始加工，到最后一个作业在机器M_2上加工完成所需的时间最少。

3.9 流水作业调度

分析：

•直观上，**一个最优调度应使机器M_1没有空闲时间，且机器M_2的空闲时间最少**。在一般情况下，机器M_2上会有机器空闲和作业积压2种情况。

•设全部作业的集合为N={1，2，…，n}。S⊆N是N的作业子集。在一般情况下，机器M1开始加工S中作业时，机器M_2还在加工其它作业，要等时间t后才可利用。将这种情况下完成S中作业所需的最短时间记为T(S,t)。流水作业调度问题的最优值为T(N,0)。

3.9 流水作业调度

设有3个作业

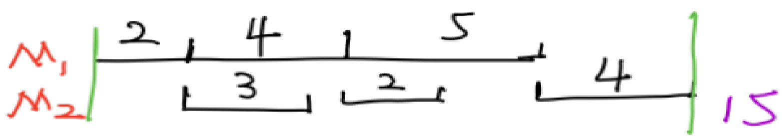
i	1	2	3
a_i	2	4	5
b_i	3	2	4

那么根据排列组合知识 按作业序号有

123，132，213，231，312，321共6种情况

选择123，321这两种情况来说明

123：

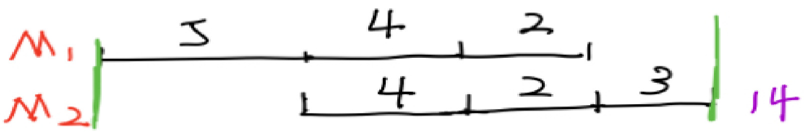


$T(\{1,2,3\},0)$

$T(\{2,3\},3)$

$T(\{3\},2)=9$

321：



$T(\{3,2,1\},0)$

$T(\{2,1\},4)$

$T(\{1\},2)=5$

流水作业调度

设π是所给n个流水作业的一个最优调度，它所需的加工时间为$a_{\pi(1)}+T'$。其中T'是在机器M_2的等待时间为$b_{\pi(1)}$时，安排作业π(2)，…，π(n)所需的时间。

记$S=N-\{\pi(1)\}$，则有$T'=T(S,b_{\pi(1)})$。

证明：事实上，由T的定义知$T'\geq T(S,b_{\pi(1)})$。若$T'>T(S,b_{\pi(1)})$，设π'是作业集S在机器M_2的等待时间为$b_{\pi(1)}$情况下的一个最优调度。则π(1)，π'(2)，…，π'(n)是N的一个调度，且该调度所需的时间为$a_{\pi(1)}+T(S,b_{\pi(1)})<a_{\pi(1)}+T'$。这与π是N的最优调度矛盾。故$T'\leq T(S,b_{\pi(1)})$。从而$T'=T(S,b_{\pi(1)})$。这就证明了流水作业调度问题具有最优子结构的性质。

由流水作业调度问题的最优子结构性质可知，

$$T(N,0)=\min_{1\leq i\leq n}\{a_i+T(N-\{i\},b_i)\}$$

$$T(S,t)=\min_{i\in S}\{a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})\}$$

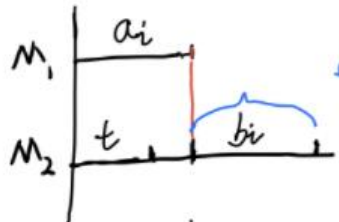
选定作业i为S中第一个加工作业之后，在机器M_2上开始对S-{i}中的作业进行加工之前，所需要的等待时间为$b_i + \max\{t-a_i,0\}$。

原因：若M_2在开始加工S中的作业之前需等待t个时间单位，且$t > a_i$，则作业i在M_1上加工完毕（需时a_i）之后，还要再等$t-a_i$个时间单位才能开始在M_2上加工；若$t \le a_i$，则作业i在M_1上加工完毕之后，立即可以在M_2上加工，等待时间为0。故M_2在开始对S-{i}中的作业进行加工之前，所需要的等待时间为$t' = b_i + \max\{t-a_i,0\}$。（$b_i$是作业i在$M_2$上加工所需的时间）。所以，假定$a_i$为已知的使得T(S,t)值最小的第一个执行的作业，可以得到

$$T(S,t)= a_i + T(S-\{i\}, b_i + \max\{t-a_i,0\})$$

$T(S,t)$：M_1开始加工S中的作业i时，M_2要等待t时后可利用

情形1：$a_i > t$

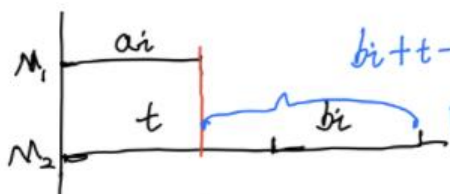


b_i

即下一个作业开始时 M_2要等b_i时间后才可以利用

则 $T(S,t) = \{a_i + T(S-\{i\}, b_i)\}$，$i \in S$ ①

情形2：$a_i < t$



$b_i + t - a_i$

即下一个作业开始 M_2要等 b_i+t-a_i时间后才可以利用

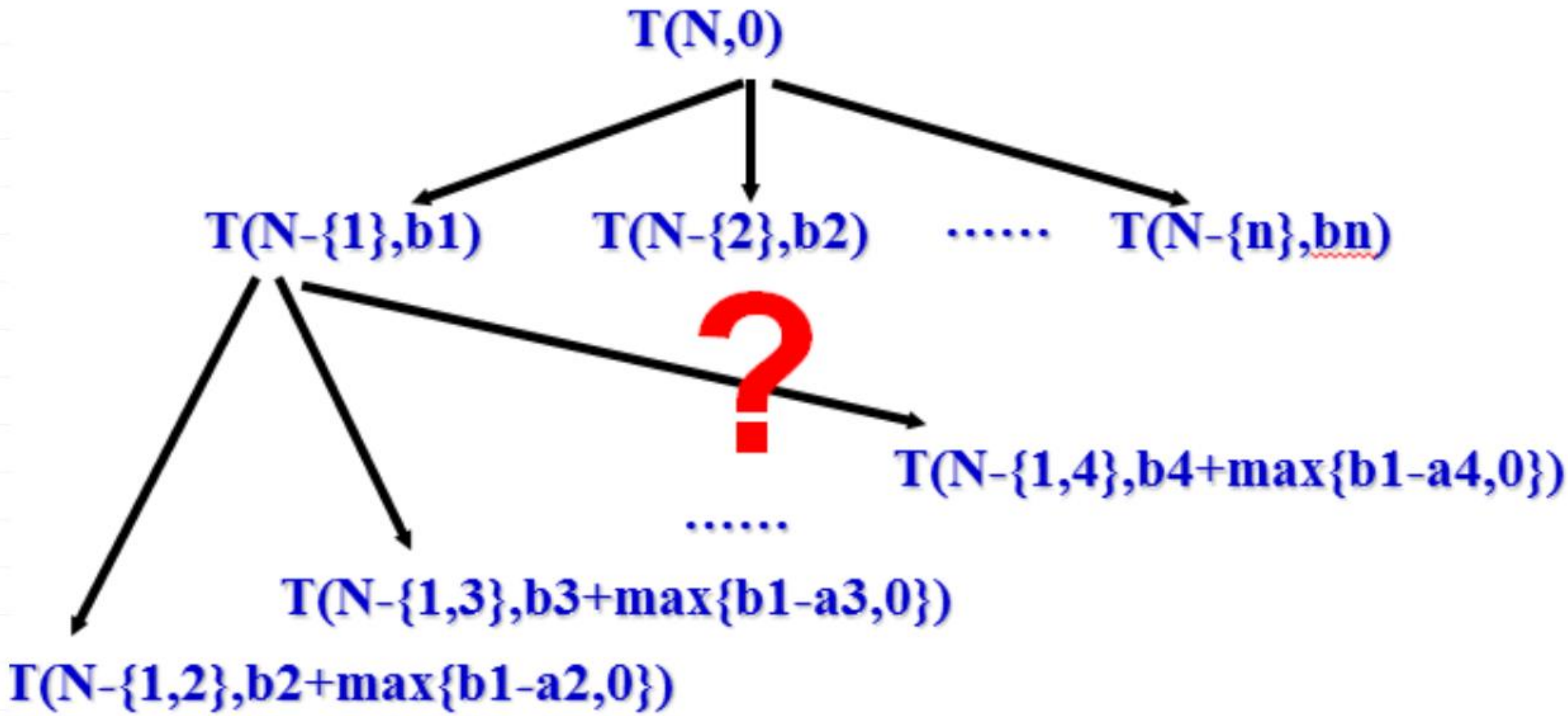
则 $T(S,t) = \{a_i + T(S-\{i\}, b_i + t - a_i)\}$ $i \in S$ ②

综合①②得

$$T(S,t) = \{a_i + T(S-\{i\}, b_i + \max(t-a_i, 0))\}$$

流水作业调度问题

- 子问题重叠性质



虽然满足最优子结构性质，也在一定程度满足子问题重叠性质。N的每个非空子集都计算一次，共2^n-1次，指数级的。

为了解决这个问题引入**Johnson不等式**

Johnson不等式

对递归式的深入分析表明，算法可进一步得到简化。

设π是作业集S在机器M_2的等待时间为t时的任一最优调度。若π(1)=i, π(2)=j。则由动态规划递归式可得:

$T(S,t)=a_i+T(S-\{i\},b_i+\max\{t-a_i,0\})=a_i+a_j+T(S-\{i,j\},t_{ij})$

其中，

$$\begin{aligned} t_{ij} &= b_j + \max\{b_i + \max\{t-a_i,0\}-a_j,0\} \\ &= b_j + b_i - a_j + \max\{\max\{t-a_i,0\},a_j-b_i\} \\ &= b_j + b_i - a_j + \max\{t-a_i,a_j-b_i,0\} \\ &= b_j + b_i - a_j - a_i + \max\{t,a_i+a_j-b_i,a_i\} \end{aligned}$$

如果作业i和j满足$\min\{b_i,a_j\}\geq\min\{b_j,a_i\}$，则称作业i和j满足**Johnson**不等式。

流水作业调度的Johnson法则

$t_{ij} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\}$

交换作业i和作业j的加工顺序，得到作业集S的另一调度，它所需的加工时间为T'(S,t)=a_i+a_j+T(S-{i,j},t_{ji})

其中，$t_{ji} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_j, a_j\}$

当作业i和j满足Johnson不等式时，有

$$\max\{-b_i, -a_j\} \le \max\{-b_j, -a_i\}$$

$$a_i + a_j + \max\{-b_i, -a_j\} \le a_i + a_j + \max\{-b_j, -a_i\}$$

$$\max\{a_i + a_j - b_i, a_i\} \le \max\{a_i + a_j - b_j, a_j\}$$

$$\max\{t, a_i + a_j - b_i, a_i\} \le \max\{t, a_i + a_j - b_j, a_j\}$$

由此可见当作业i和作业j满足Johnson不等式时，交换它们的加工顺序后，会增加加工时间。对于流水作业调度问题，必存在最优调度π，使得任意i，作业π(i)和π(i+1)满足Johnson不等式。进一步还可以证明，调度满足Johnson法则当且仅当对任意i<j有

$$\min\{b_{\pi(i)}, a_{\pi(j)}\} \ge \min\{b_{\pi(j)}, a_{\pi(i)}\}$$

由此可知，**所有满足Johnson法则的调度均为最优调度。**

算法描述

流水作业调度问题的Johnson算法

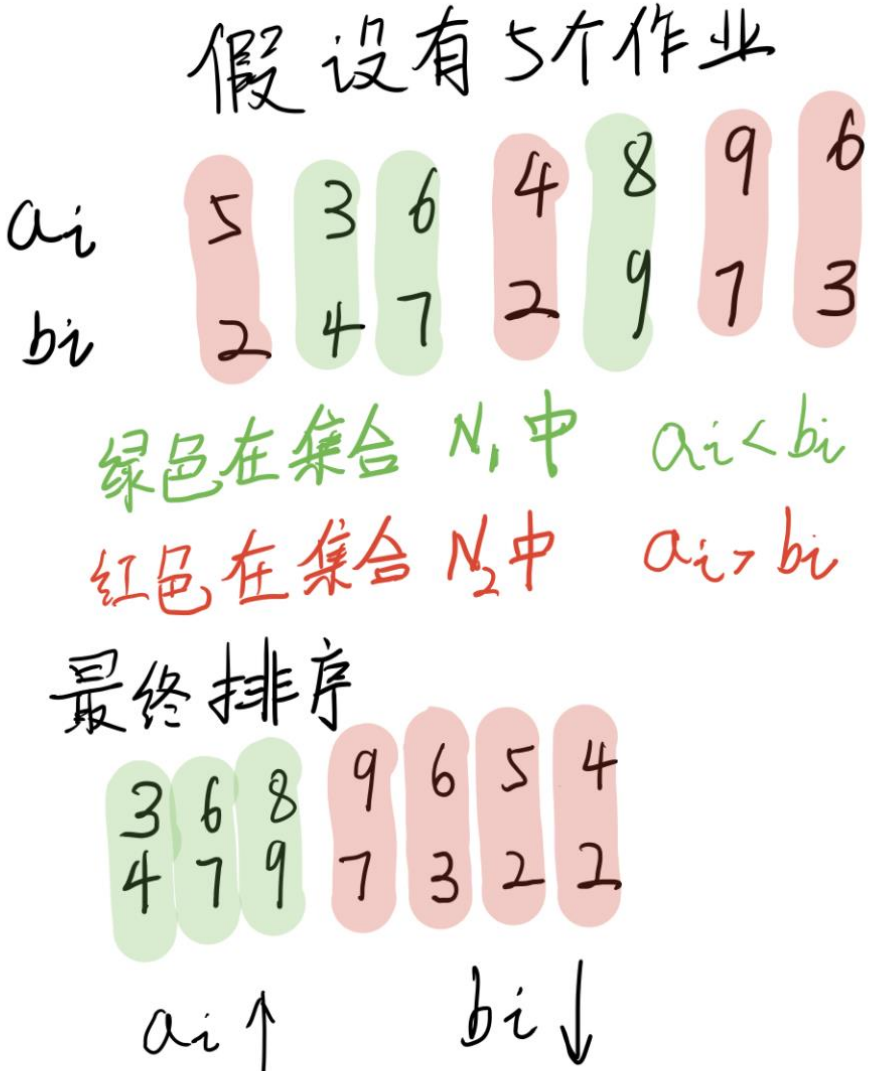
(1)令 $N_1 = \{i \mid a_i < b_i\}, N_2 = \{i \mid a_i \ge b_i\};$

(2)将N_1中作业依a_i的非减序排序；将N_2中作业依b_i的非增序排序；

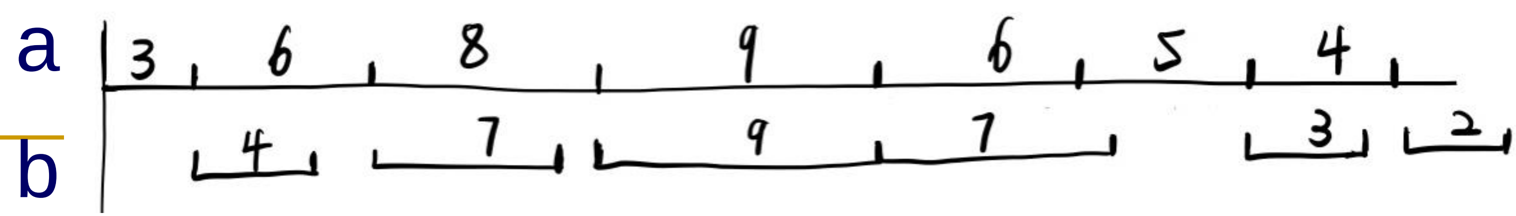
(3)N_1中作业接N_2中作业构成满足Johnson法则的最优调度。

算法复杂度分析：

算法的主要计算时间花在对作业集的排序。因此，在最坏情况下算法所需的计算时间为O(nlogn)。所需的空间为O(n)。



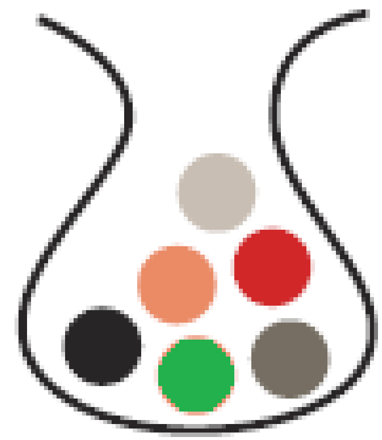
- 推测一下这个Johson法则为什么能够得到最小的作业时间?
- Johson法则分出的第一组都是b加工时间大于a的，且按a时间递增；分出的第二组都是a加工时间大于b的，且按b时间递减。
- 由于a加工是无间断的，决定时间长短的只是b。按照Johson法则会发现，中间部分都是一些b耗时大的作业，两头都是一些耗时小的作业，这样安排会很好填充b中的时间空隙。



3.10 0-1背包问题

给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个特殊的整数规划问题。



$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

0-1背包问题

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优值为m(i，j)，即m(i，j)是背包容量为j，可选择物品为i，i+1，…，n时0-1背包问题的最优值。由0-1背包问题的最优子结构性质，可以建立计算m(i，j)的递归式如下。

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1, j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$

- 若背包剩余容量大于第i个物品重量，则分别考虑装与不装两种情况的重量，取最大值
- 若背包剩余容量小于第i个物品重量，则不考虑这个物品

$$m(n,j) = \begin{cases} v_n & j \ge w_n \\ 0 & 0 \le j < w_n \end{cases}$$

```
void Knapsack(int v[],int w[],int c,int n,int m[][10])
{
    int jMax = min(w[n]-1,c);//背包剩余容量上限 范围[0~w[n]-1]
    for(int j=0; j<=jMax;j++)
    {
        m[n][j]=0;
    }

    for(int j=w[n]; j<=c; j++)//限制范围[w[n]~c]
    {
        m[n][j] = v[n];
    }

    for(int i=n-1; i>1; i--)
    {
        jMax = min(w[i]-1,c);
        for(int j=0; j<=jMax; j++)//背包不同剩余容量j<=jMax<c
        {
            m[i][j] = m[i+1][j];//没产生任何效益
        }
        for(int j=w[i]; j<=c; j++) //背包不同剩余容量j-wi >c
        {
            m[i][j] = max(m[i+1][j],m[i+1][j-w[i]]+v[i]);//效益值增长vi
        }
    }
    m[1][c] = m[2][c];
    if(c>=w[1])
    {
        m[1][c] = max(m[1][c],m[2][c-w[1]]+v[1]);
    }
}
```

```
//x[]数组存储对应物品0-1向量,0不装入背包，1表示装入背包
void Traceback(int m[][10],int w[],int c,int n,int x[])
{
    for(int i=1; i<n; i++)
    {
        if(m[i][c] == m[i+1][c])
        {
            x[i]=0;
        }
        else
        {
            x[i]=1;
            c-=w[i];
        }
    }
    x[n]=(m[n][c])?1:0;
}
```

n= 4 c=8

w[]={1,4,2,3} v[]={2,1,4,3}

j / i	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8
i=4	0	0	3	3	3	3	3	3
i=3	0	4	4	4	7	7	7	7
i=2	0	4	4	4	7	7	7	7
i=1								9

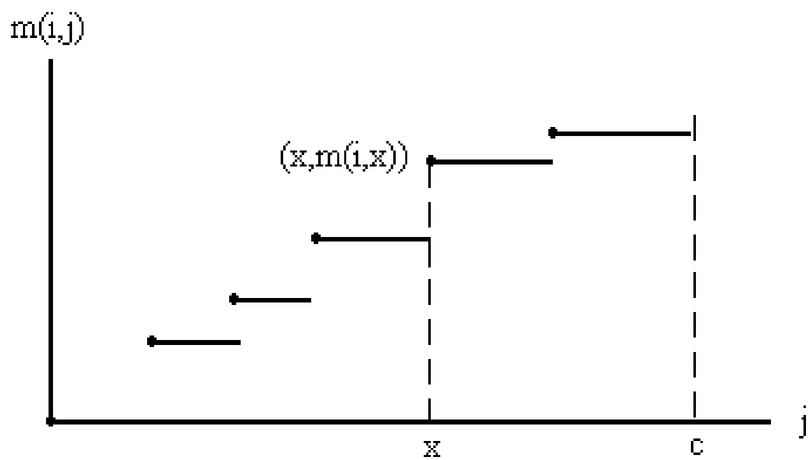
绿色框为方法 Traceback 的回溯过程 x[]={1,0,1,1}

算法复杂度分析：

从Knapsack容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

算法改进

由m(i,j)的递归式容易证明，在一般情况下，对每一个确定的i(1≤i≤n)，函数m(i,j)是关于变量j的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数m(i,j)由其全部跳跃点唯一确定。如图所示。



对每一个确定的i(1≤i≤n)，用一个表p[i]存储函数m(i，j)的全部跳跃点。表p[i]可根据计算m(i，j)的递归式来递归地由表p[i+1]计算，初始时p[n+1]={(0，0)}。

算法改进

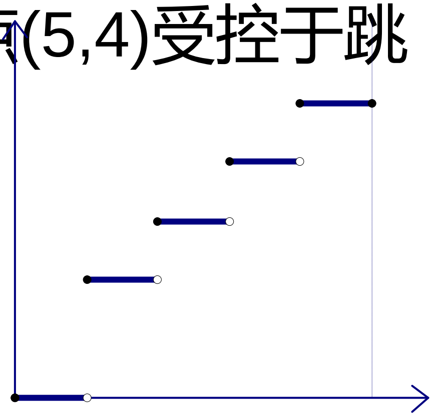
$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j\geq w_i\\ m(i+1,j) & 0\leq j<w_i\end{cases}$$

- 函数m(i,j)是由函数m(i+1,j)与函数m(i+1,j-wi)+vi作max运算得到的。因此，函数m(i,j)的全部跳跃点包含于函数m(i+1，j)的跳跃点集p[i+1]与函数m(i+1，j-wi)+vi的跳跃点集q[i+1]的并集中。易知，(s,t)∈q[i+1]当且仅当wi≤s≤c且(s-wi,t-vi)∈p[i+1]。因此，容易由p[i+1]确定跳跃点集q[i+1]如下
q[i+1]=p[i+1]⊕(wi,vi)={(j+wi,m(i,j)+vi)|(j,m(i,j))∈p[i+1]}
- 另一方面，设(a，b)和(c，d)是p[i+1]∪q[i+1]中的2个跳跃点，则当c≥a且d<b时，(c，d)受控于(a，b)，从而(c，d)不是p[i]中的跳跃点。除受控跳跃点外，p[i+1]∪q[i+1]中的其它跳跃点均为p[i]中的跳跃点。
- 由此可见，在递归地由表p[i+1]计算表p[i]时，可先由p[i+1]计算出q[i+1]，然后合并表p[i+1]和表q[i+1]，并清除其中的受控跳跃点得到表p[i]。

一个例子

n=5，c=10，w={2，2，6，5，4}，v={6，3，5，4，6}。

初始时p[6]={(0,0)}，(w5,v5)=(4,6)。因此，
q[6]=p[6]⊕(w5,v5)={(4,6)}。
p[5]={(0,0),(4,6)}。
q[5]=p[5]⊕(w4,v4)={(5,4),(9,10)}。从跳跃点集p[5]与q[5]的并集p[5]∪q[5]={(0,0),(4,6),(5,4),(9,10)}中看到跳跃点(5,4)受控于跳跃点(4,6)。将受控跳跃点(5,4)清除后，得到
p[4]={(0,0),(4,6),(9,10)}
q[4]=p[4]⊕(6，5)={(6，5)，(10，11)}
p[3]={(0，0)，(4，6)，(9，10)，(10，11)}
q[3]=p[3]⊕(2，3)={(2，3)，(6，9)}
p[2]={(0，0)，(2，3)，(4，6)，(6，9)，(9，10)，(10，11)}
q[2]=p[2]⊕(2，6)={(2，6)，(4，9)，(6，12)，(8，15)}
p[1]={(0，0)，(2，6)，(4，9)，(6，12)，(8，15)}
p[1]的最后的那个跳跃点(8,15)给出所求的最优值为m(1,c)=15。



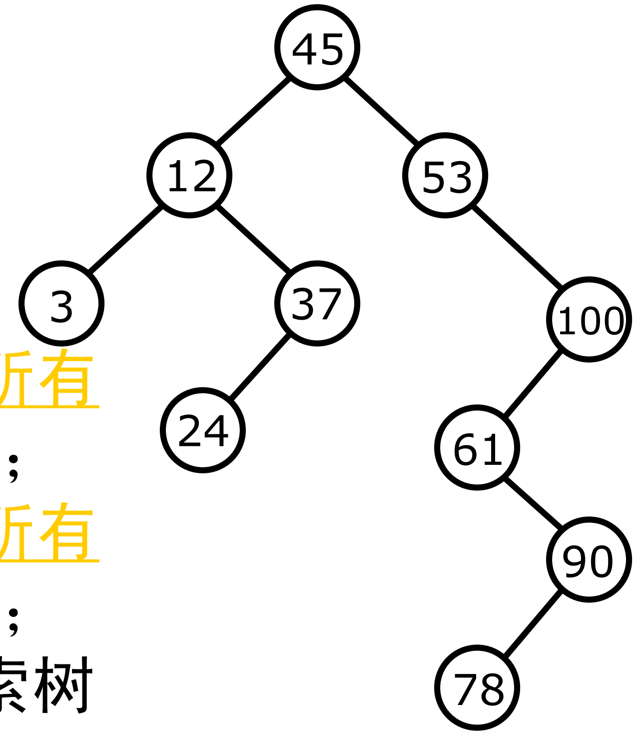
算法复杂度分析

上述算法的主要计算量在于计算跳跃点集p[i]($1 \le i \le n$)。由于$q[i+1]=p[i+1] \oplus (w_i, v_i)$，故计算q[i+1]需要O(|p[i+1]|)计算时间。合并p[i+1]和q[i+1]并清除受控跳跃点也需要O(|p[i+1]|)计算时间。从跳跃点集p[i]的定义可以看出，p[i]中的跳跃点相应于$x_i,\ldots,x_n$的0/1赋值。因此，p[i]中跳跃点个数不超过2^{n-i+1}。由此可见，算法计算跳跃点集p[i]所花费的计算时间为 $O\left(\sum_{i=2}^{n}|p[i+1]|\right)=O\left(\sum_{i=2}^{n}2^{n-i}\right)=O(2^n)$

从而，改进后算法的计算时间复杂性为$O(2^n)$。当所给物品的重量w_i($1 \le i \le n$)是整数时，$|p[i]| \le c+1$，($1 \le i \le n$)。在这种情况下，改进后算法的计算时间复杂性为$O(\min\{nc, 2^n\})$。

3.11 最优二叉搜索树

- 二叉搜索树



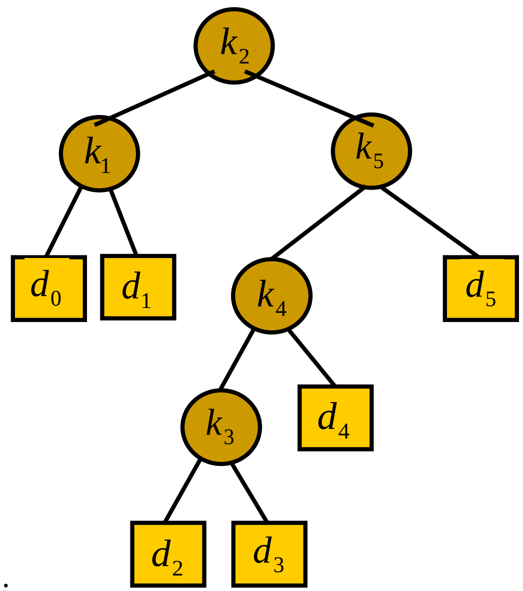
（1）若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

（2）若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

（3 它的左、右子树也分别为二叉搜索树

在随机的情况下，二叉查找树的平均查找长度和$\log n$是等数量级的

二叉查找树的期望耗费



- 查找成功与不成功的概率

$$\sum_{i=1}^{n} a_i + \sum_{i=0}^{n} b_i = 1$$

- 二叉查找树的期望耗费（平均路长）

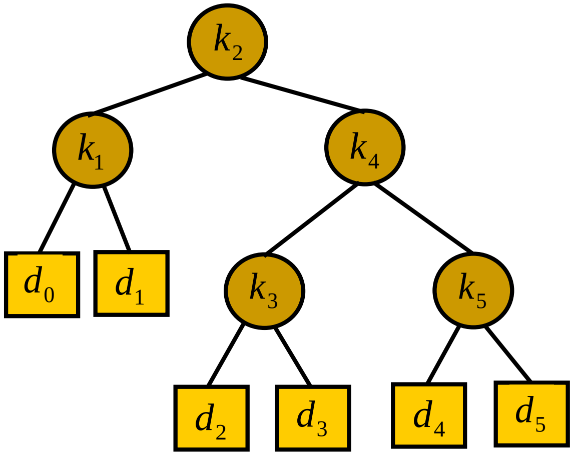
$$p = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot b_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot a_i$$

$$w_{i,j} = a_{i-1} + b_i + \cdots + b_j + a_j$$

$$w_{i,j} p_{i,j} = w_{i,j} + w_{i,m-1} p_l + w_{m+1,j} p_r$$

- 有 n 个节点的二叉树的个数为：
  穷举搜索法的时间复杂度为指数级 $\Omega(4^n / n^{3/2})$

二叉查找树的期望耗费示例



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.10
d_1	2	0.10	0.20
d_2	3	0.05	0.15
d_3	3	0.05	0.15
d_4	3	0.05	0.15
d_5	3	0.10	0.30
Total			2.40

最优二叉搜索树

$$w_{i,j} = a_{i-1} + b_i + \cdots + b_j + a_j$$

最优二叉搜索树T_{ij}的平均路长为p_{ij}，则所求的最优值为$p_{1,n}$。由最优二叉搜索树问题的最优子结构性质可建立计算p_{ij}的递归式如下

$$w_{i,j} p_{i,j} = w_{i,j} + \min_{i \le k \le j} \{ w_{i,k-1} p_{i,k-1} + w_{k+1,j} p_{k+1,j} \}$$

记$w_{i,j}p_{i,j}$为$m(i,j)$，则$m(1,n)=w_{1,n}p_{1,n}=p_{1,n}$为所求的最优值。计算$m(i,j)$的递归式为

$$m(i,j) = w_{i,j} + \min_{i \le k \le j} \{ m(i,k-1) + m(k+1,j) \}, \quad i \le j$$

$$m(i,i-1) = 0, \quad 1 \le i \le n$$

注意到，

$$\min_{i \le k \le j} \{ m(i,k-1) + m(k+1,j) \} = \min_{s[i][j-1] \le k \le s[i+1][j]} \{ m(i,k-1) + m(k+1,j) \}$$

可以得到$O(n^2)$的算法

- 课后作业1：
- 请用动态规划的方法进行代码实验。

https://leetcode.com/problems/unique-binary-search-trees/

思路：给定一个有序序列 1…n，为了构建出一棵二叉搜索树，我们可以遍历每个数字 i，将该数字作为树根，将 1…(i−1) 序列作为左子树，将 (i+1)…n 序列作为右子树。接着我们可以按照同样的方式递归构建左子树和右子树。在上述构建的过程中，由于根的值不同，因此我们能保证每棵二叉搜索树是唯一的。由此可见，原问题可以分解成规模较小的两个子问题，且子问题的解可以复用。因此，我们可以使用动态规划来求解本题。

- 课后作业2：
- 书本算法实现题：3-1，3-10

下下个周三前发送至邮箱 dawei_course@163.com，作业以 “学号_姓名_算法第3次作业” 为主题命名邮件，附件名 “2010110_张三_算法第三次作业.pdf” 所有题目解答合并为一个pdf文件