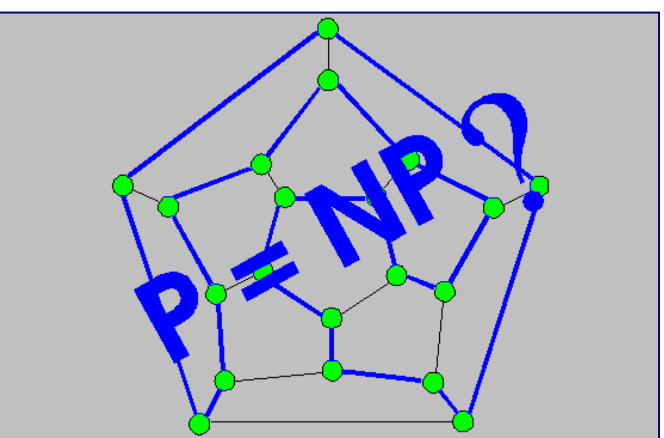
5-动态规划

陆伟

算法设计与分析

Introduction to the Design and Analysis of Algorithms



October 15, 2022

Lecture Overview

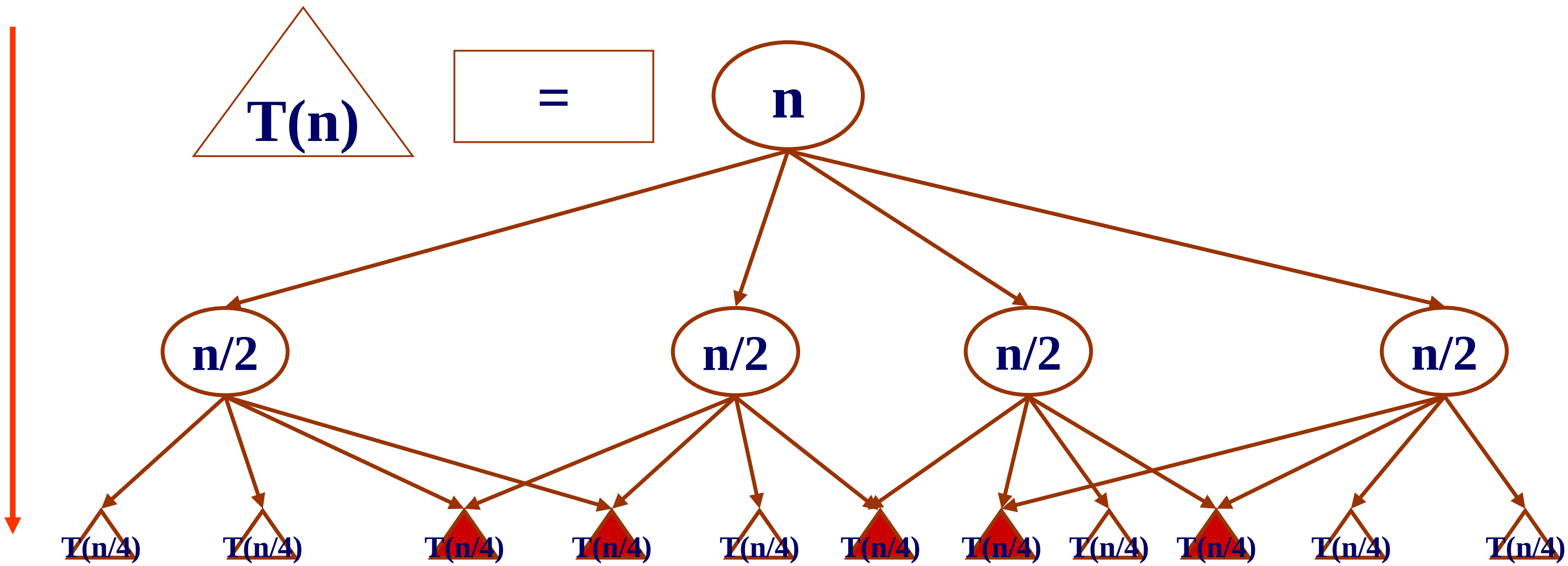
1. 基本思想
2. 适用条件
3. 基本步骤与要素
4. 典型案例
5. 开放性讨论
6. 总结

基本思想

- 动态规划是一种使多阶段决策过程最优的通用方法。
- 动态规划算法与分治法类似，其思想把求解的问题分成**许多阶段**或**多个子问题**，然后按顺序求解各子问题。最后一个阶段或子问题的解就是初始问题的解。

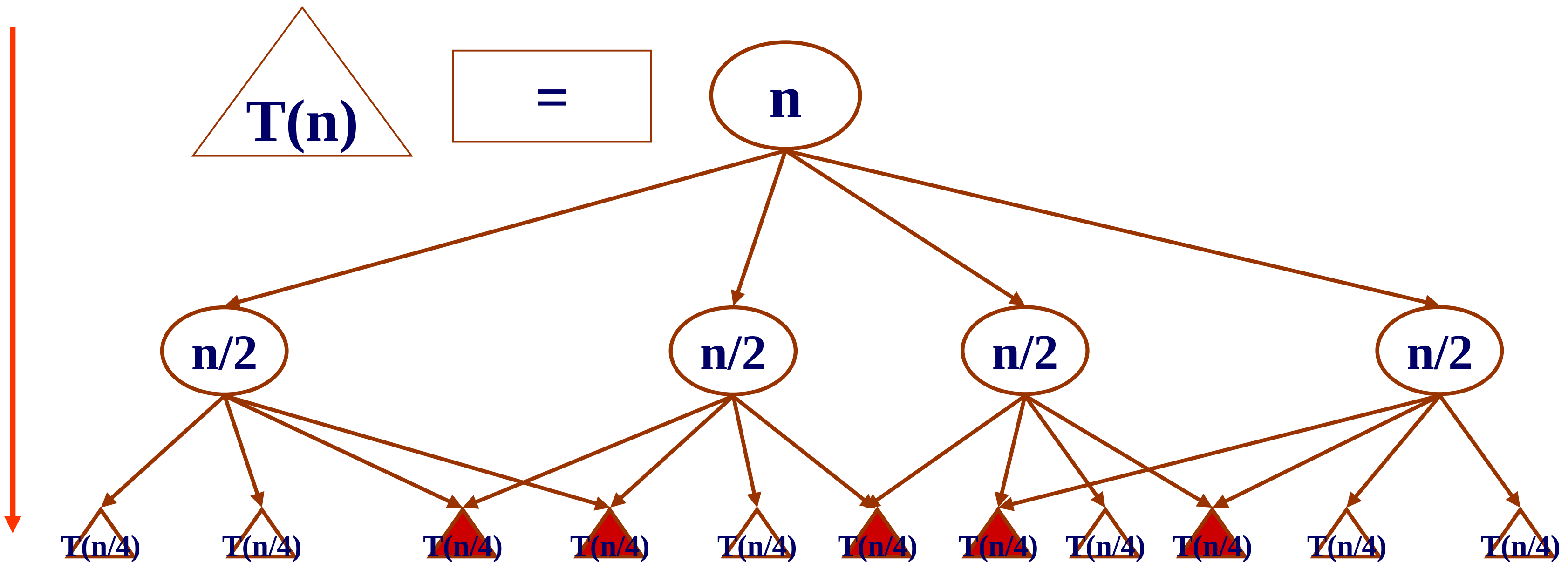
基本思想

- 动态规划中分解得到的子问题往往不是互相独立的。但不同子问题的数目常常只有多项式级。用分治法求解时，有些子问题被重复计算了许多次，从而导致分治法求解问题时间复杂度较高。



基本思想

- 动态规划基本思想是**保留**已解决的子问题的解，在需要时再**查找**已求得的解，就可以**避免大量重复计算**，进而提升算法效率。



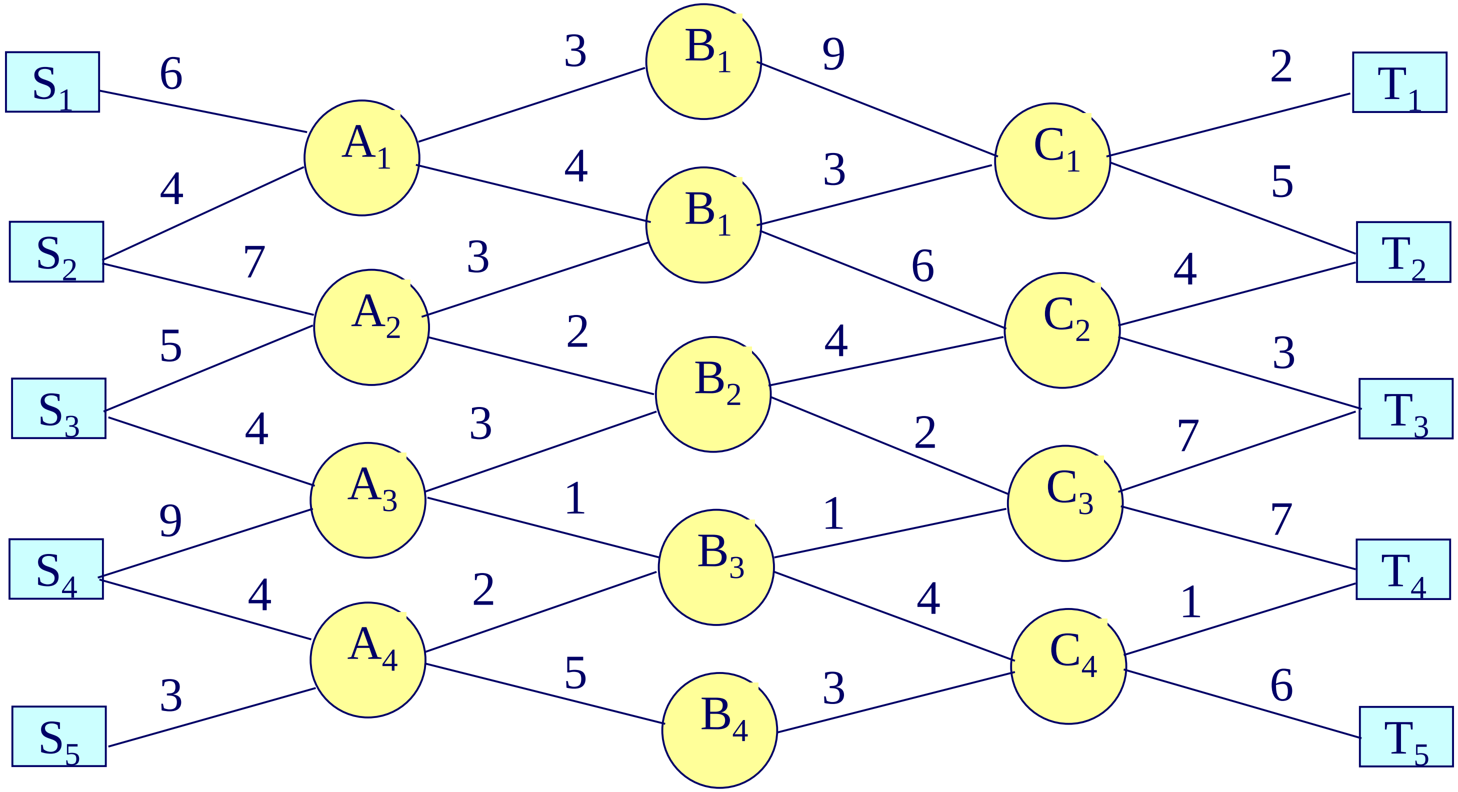
适用条件

- 动态规划问题的特征
  - 求解的问题是组合优化问题
  - 求解过程需要多步判断，从小到大依次求解
  - 子问题目标函数最优解之间存在依赖关系（原问题最优解是由子问题最优解构成）

适用条件

- 多起点、多终点的最短路径

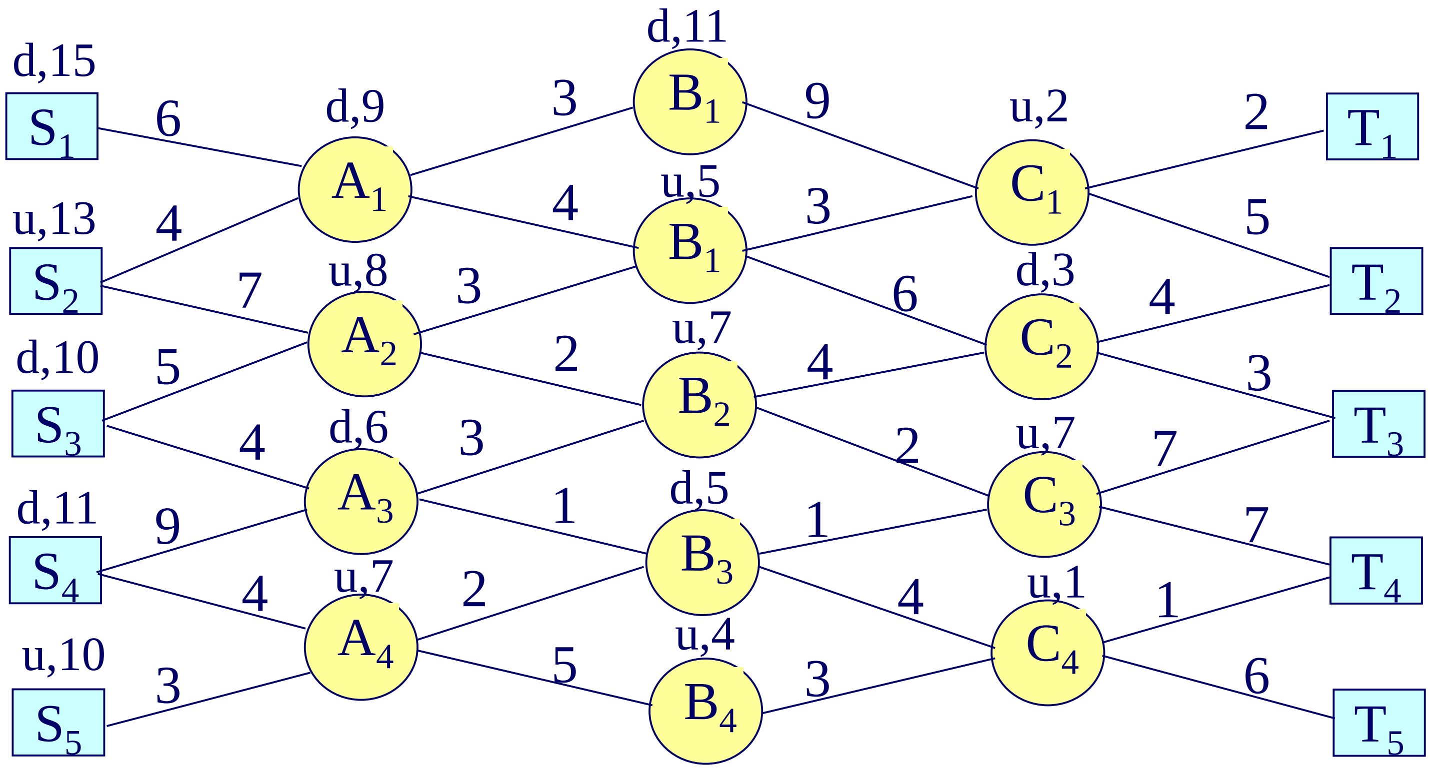
讨论与理解



适用条件

- 多起点、多终点的最短路径

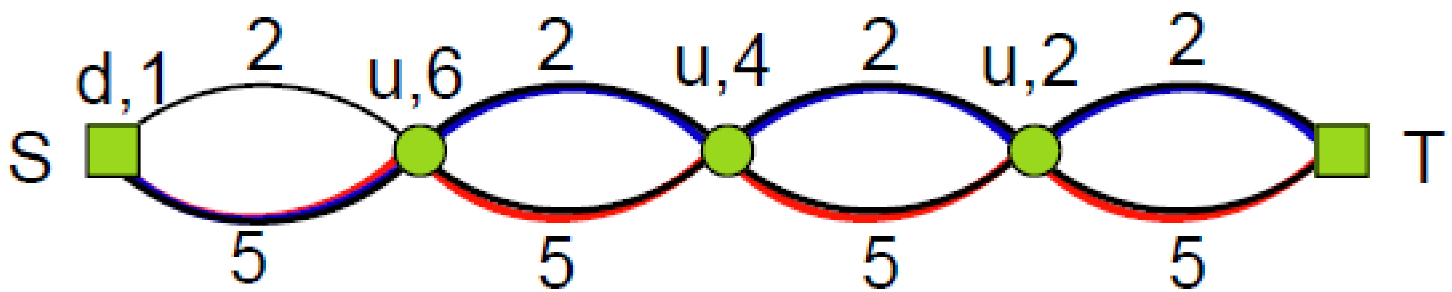
讨论与理解



适用条件

- 求图中总长模10的最短路径

讨论与理解



基本步骤与要素

- 基本步骤

  （1）找出最优解的性质，并刻画其结构特征。

  （2）递推地定义最优值。

  （3）以自底向上的方式计算出最优值。

  （4）根据计算最优值时得到的信息，构造最优解。

- 要素
  - 最优子结构
  - 重叠子问题
  - 备忘录（表格）

典型案例

- ## 矩阵连乘问题

问题：给定n个矩阵$\{A_1, A_2, \ldots, A_n\}$，其中，$A_i$与$A_{i+1}$是可乘的，$i=1, 2, \ldots, n-1$。确定矩阵乘法顺序，使得元素相乘的次数最少。

实例：3个矩阵$\{A_1, A_2, A_3\}$连乘，设3个矩阵的维数分别为10×100, 100×5, 5×50。若按((A1A2)A3)计算，需要数乘次数为：

$10\times100\times5+10\times5\times50=7500$;

若按(A1(A2A3))计算，需要数乘次数为：

$100\times5\times50+10\times100\times50=75000$。

矩阵连乘次序对计算量有很大影响。

典型案例

$p_0p_1 \times p_1p_2 \times \dots \times p_{k-1}p_k \times \dots \times p_{n-1}p_n$
$A1 \times A2 \times \dots \times Ak \times \dots \times An$

分析

- 矩阵连乘问题

蛮力法：搜索所有可能的计算次序，并计算出每种计算次序相应需要的数乘次数，从中找出一种数乘次数最少的计算次序。设不同计算次序为P(n)。

复杂度分析

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

该递归方程解为Catalan数（ $C(n)=\frac{1}{n+1}\binom{2n}{n}$ ）

$P(n)=\Omega(4^n/n^{3/2})$

$$p_0p_1 \times p_1p_2 \times \ldots \times p_{k-1}p_k \times \ldots \times p_{n-1}p_n$$
$$A1 \times A2 \times \ldots \times Ak \times \ldots \times An$$

典型案例

- 矩阵连乘问题—问题理解

说明:
将矩阵连乘积$A_iA_{i+1}\ldots A_j$简记为A[i:j]，$i\le j$。
考察计算A[i:j]的最优计算次序。设这个计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，$i\le k<j$，则其相应完全加括号方式为$(A_iA_{i+1}\ldots A_K)(A_{k+1}A_{k+2}\ldots A_j)$
计算量:
A[i:k]的计算量加上A[k+1:j]的计算量，再加上A[i:k]和A[k+1:j]相乘的计算量。

典型案例

- 矩阵连乘问题—动态规划

（1）分析最优解结构:

➢计算A[i:j]的最优次序所包含的计算矩阵子链 A[i:k]和A[k+1:j]的次序也是最优的。

➢矩阵连乘计算次序问题的最优解包含着其子问题的最优解，满足最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法求解的显著特征。

典型案例

- 矩阵连乘问题—动态规划

（2）建立递推关系

1.设计算A[i:j]，$1 \le i \le j \le n$，所需要的最少数乘次数m[i,j]，则原问题的最优值为m[1,n]。

2.递推方程

$$m[i,j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\} & i<j \end{cases}$$

推导

K的位置只有j-i种可能

典型案例

$p_0p_1 \times p_1p_2 \times \ldots \times p_{k-1}p_k \times \ldots \times p_{n-1}p_n$

$A1 \times A2 \times \ldots \times Ak \times \ldots \times An$

S[i][j]表示矩阵链A[i:j]最少乘次的断开位置。

- 矩阵连乘问题—动态规划

（3）计算最优值—递归求解

```
int RecurMatrixChain(int i, int j, int *p, int **s) {
   if(i == j) return 0;
   int u = RecurMatrixChain(i, i) + RecurMatrixChain(i+1, j)  + p[i-1]*p[i]*p[j];
   s[i][j] = i;
   for(int k = i+1; k < j; k++){
      int t = RecurMatrixChain(i, k) + RecurMatrixChain(k+1, j)  + p[i-1]*p[k]*p[j];
      if(t < u) {u = t; s[i][j] = k;}
   }
   return u;
}
```

分析

$$T(n) \geq \begin{cases} O(1) & n=1 \\ \sum_{k=1}^{n-1}(T(k)+T(n-k)+O(1)) & n>1 \end{cases}$$

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

$$T(n) \geq 2^{n-1}$$

典型案例

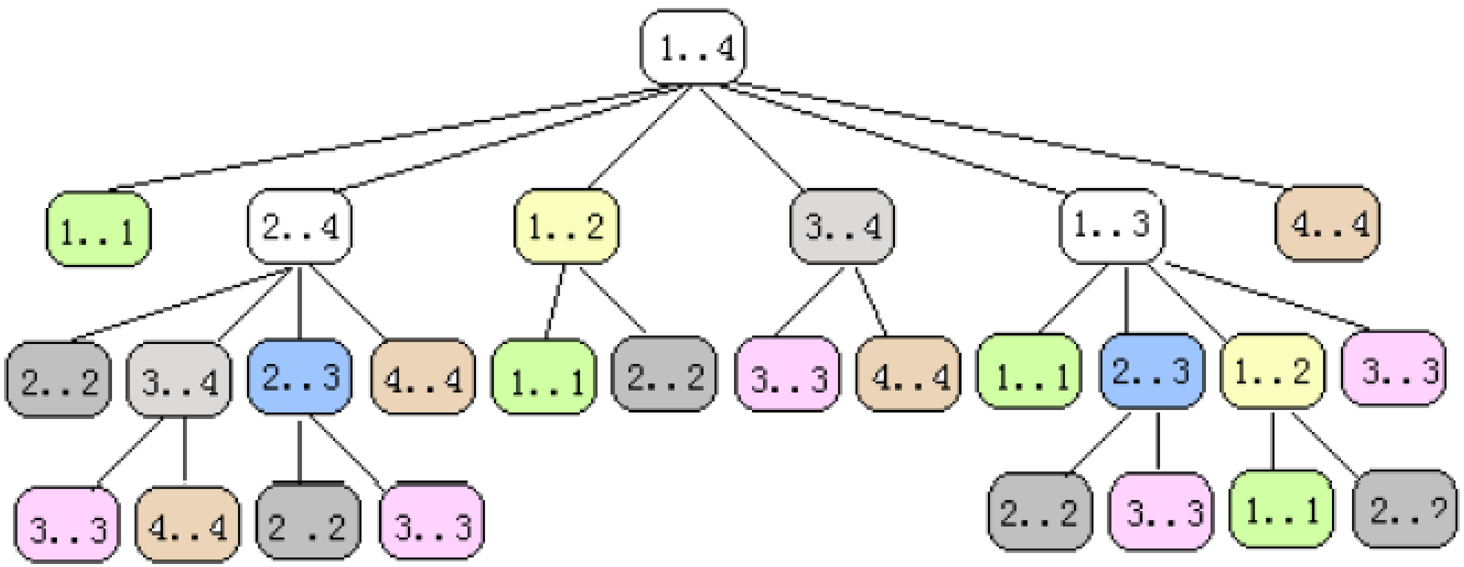
分析讨论

- 矩阵连乘问题—递归算法分析

对于$1\leq i\leq j\leq n$不同的有序对(i,j)对应于不同的子问题,不同子问题的个数最多只有:

$$\binom{n}{2}+n=\Theta(n^2)$$

递归求解最优值复杂度较高的原因是：子问题重复度高

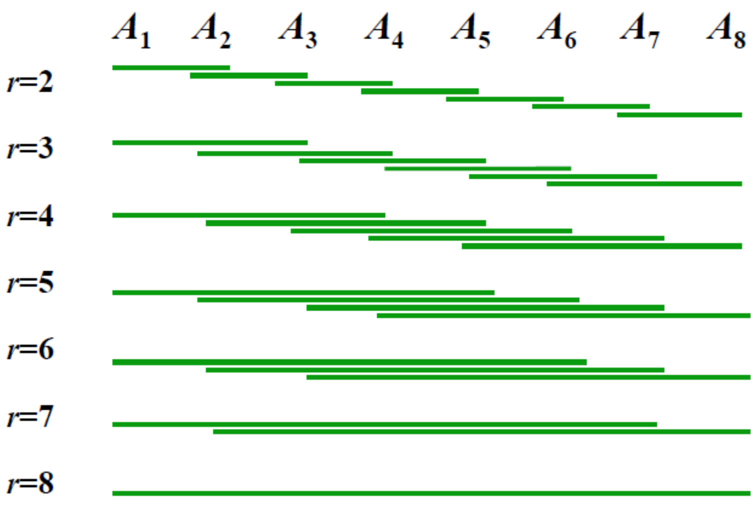


典型案例

- 矩阵连乘问题—程序设计 实践

（3）计算最优值—迭代查表求解

```
void MatrixChain(int *p，int n，int **m，int **s) {
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
        for (int i = 1; i <= n-r+1; i++) {
            int j = i+r-1;
            m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
            s[i][j] = i;
            for (int k = i+1; k < j; k++) {
                int t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (t < m[i][j]) { m[i][j] = t; s[i][j] = k;}
            }
        }
}
```



时间复杂度$O(n^3)$
空间复杂的 $O(n^2)$

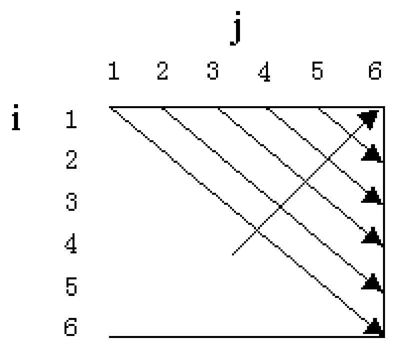
典型案例

例

- 矩阵连乘问题—实例分析

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5 = 0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5 = 2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5 = 4375+0+35\times10\times20=11375 \end{cases}$$



(a) 计算次序

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

典型案例

- 矩阵连乘问题—程序设计

（4）构造最优解

```
void Tracback (int i, int j, int **s) {
    if (i == j) return;
    Trackback (i, s[i][j], s);
    Trackback (s[i][j] + 1, j, s);
    cout << “Multiply” << i << “,” <<s[i][j];
    cout << “and A” << (s[i][j]) + 1 << “,” << j << endl;
}
```

典型案例

例

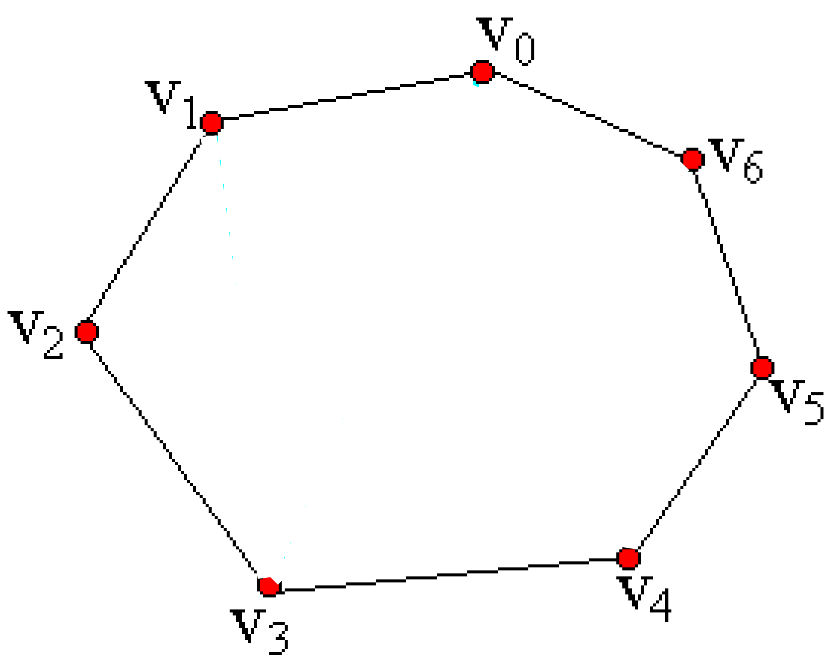
- 矩阵连乘问题—实例计算

设输入P=<30, 35, 15, 5,10,20>， n=5，相应矩阵链为:
A_1, A_2, A_3, A_4, A_5，其中，
A_1:30×35, A_2:35×15, A_3:15×5, A_4:5×10, A_5:10×20

典型案例

- 凸多边形最优三角剖分

凸多边形的表示：多边形顶点的逆时针序列表示。
例：$P=\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$

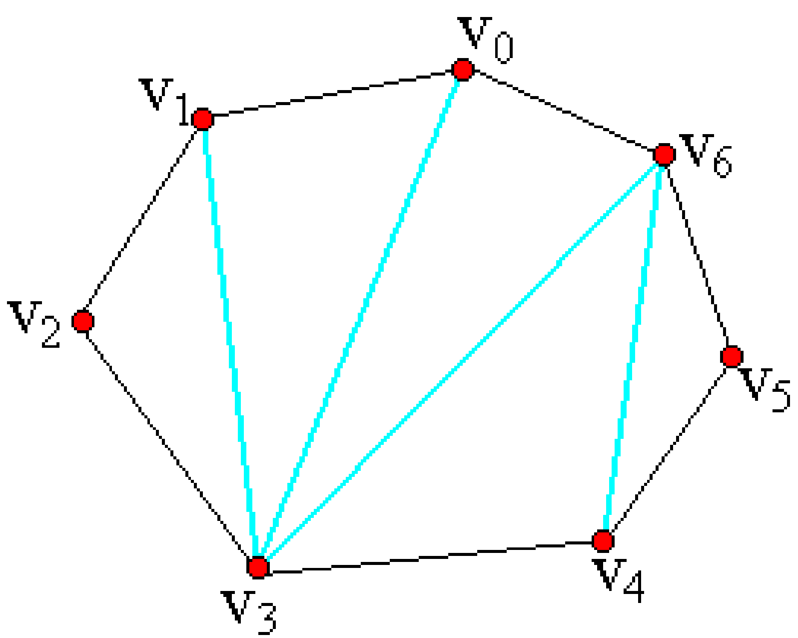


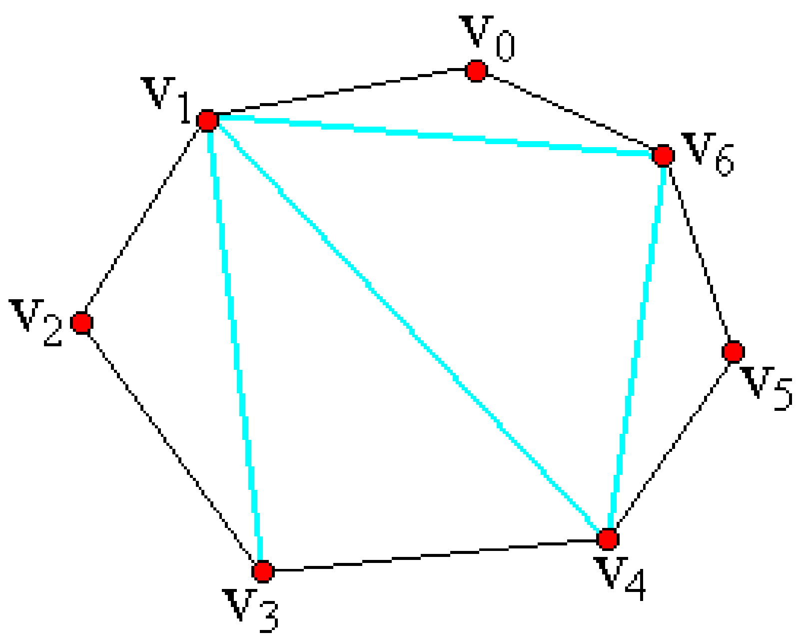
弦：两个不相邻顶点间的直线线段。弦将多边形分割为两个子多边形。

典型案例

- 凸多边形最优三角剖分

多边形的三角剖分指将多边形分割成互不相交的三角形的弦的集合T。n个顶点的凸多边形剖分必有n-3条弦和n-2个三角形。





典型案例

- **凸多边形最优三角剖分**

问题：给定凸多边形$P=\{v_0, v_1, \ldots, v_{n-1}\}$，以及定义在由凸多边形的边和弦组成的三角形上的权函数w。要求确定该凸多边形的三角剖分，使得该三角剖分所对应的权，即三角剖分中诸三角形上权之和为最小。

典型案例

- **凸多边形最优三角剖分**

（1）一个表达式的完全加括号方式相应于一棵完全二叉树，称为表达式的语法树。例如，完全加括号的矩阵连乘积$((A_1(A_2A_3))(A_4(A_5A_6)))$所相应的语法树如图 (a)所示。

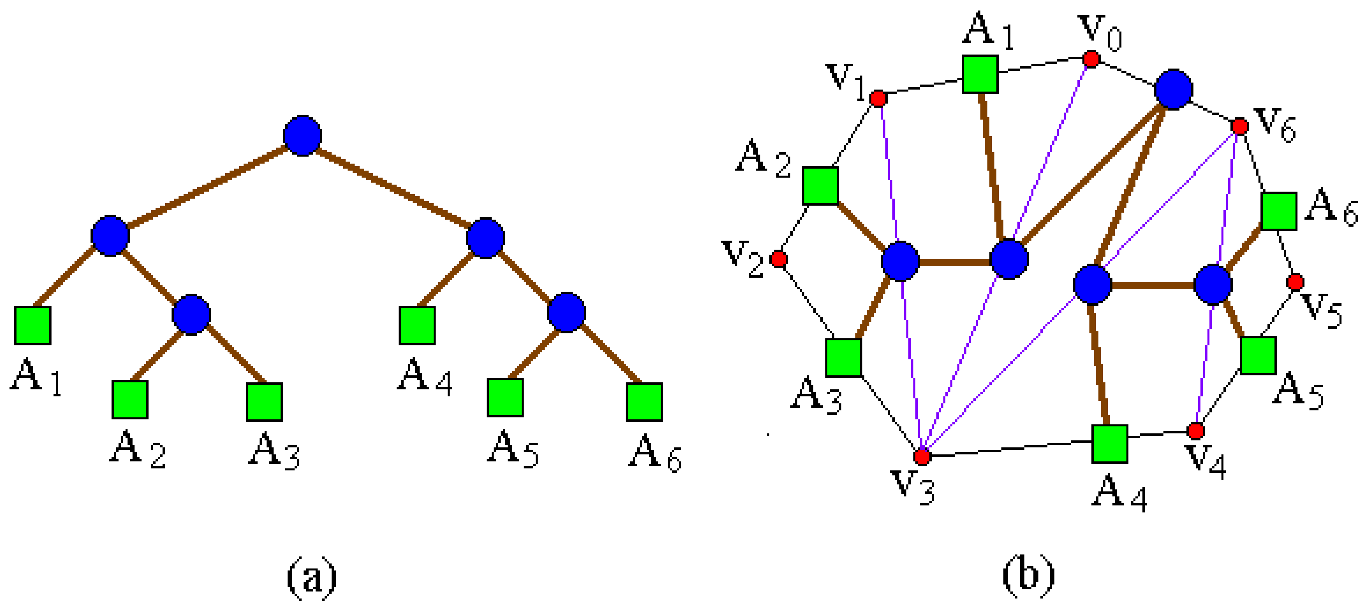
（2）凸多边形$\{v_0,v_1,\ldots v_{n-1}\}$的三角剖分也可以用语法树表示。例如，图 (b)中凸多边形的三角剖分可用图 (a)所示的语法树表示。

（3）矩阵连乘积中的每个矩阵A_i对应于凸(n+1)边形中的一条边$v_{i-1}v_i$。三角剖分中的一条弦v_iv_j，$i<j$，对应于矩阵连乘积$A[i+1:j]$。

（4）矩阵连乘的最优计算次序问题是凸多边形最优三角剖分问题的特殊情形。

典型案例

- 三角剖分的结构及其相关问题—同构



典型案例

- 凸多边形最优三角剖分

凸多边形的最优三角剖分问题有最优子结构性质。

若凸(n+1)边形P={v_0,v_1,…,v_n}的最优三角剖分T包含三角形$v_0v_kv_n$，$1\le k\le n-1$，则T的权为3个部分权的和：三角形$v_0v_kv_n$的权，子多边形{v_0,v_1,…,v_k}和{v_k,v_{k+1},…,v_n}的权之和。

由T所确定的这2个子多边形的三角剖分也是最优的。

因为若有{v_0,v_1,…,v_k}或{v_k,v_{k+1},…,v_n}的更小权的三角剖分将导致T不是最优三角剖分的矛盾。

典型案例

- 凸多边形最优三角剖分—递归结构

定义t[i][j]，$1 \le i < j \le n$为凸子多边形$\{v_{i-1}, v_i, \ldots, v_j\}$的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形$\{v_{i-1}, v_i\}$具有权值0。据此定义，要计算的凸(n+1)边形P的最优权值为t[1][n]。

t[i][j]的值可以利用最优子结构性质递归地计算。当$j-i \ge 1$时，凸子多边形至少有3个顶点。由最优子结构性质，t[i][j]的值应为t[i][k]的值加上t[k+1][j]的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中$i \le k \le j-1$。由于在计算时还不知道k的确切位置，而k的所有可能位置只有j-i个，因此可以在这j-i个位置中选出使t[i][j]值达到最小的位置。

典型案例

分析讨论

- 凸多边形最优三角剖分—递归结构

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

对比矩阵乘法

$$m[i,j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & i<j\end{cases}$$

典型案例

理解

- 最长公共子序列

若给定序列$X=\{x_1,x_2,\ldots,x_m\}$，则另一序列$Z=\{z_1,z_2,\ldots,z_k\}$是X的子序列是指，存在一个严格递增下标序列$\{i_1,i_2,\ldots,i_k\}$使得对于所有$j=1,2,\ldots,k$有：$z_j=x_{ij}$。

给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的**公共子序列**。

Z={B，C，D，B}是序列X={A，B，C，B，D，A，B}的子序列，相应的递增下标序列为{2，3，5，7}

X={A，B，C，B，D，A，B}，Y={B，D，C，A，B，A}，则序列Z={B，C，A}是X和Y的一个公共子序列。

序列W={B，C，B，A}是X和Y的一个公共子序列，并且是X和Y的最长公共子序列。

典型案例

分析讨论

- 最长公共子序列

问题：给定2个序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$，找出X和Y的最长公共子序列。

蛮力法：求X和Y的所有公共子序列，找出最长的。
判断X一个子序列是否是Y子序列时间$O(n)$。
X有2^m个子序列。
最坏情况下时间复杂度$O(n2^m)$。

典型案例

分析讨论

- 最长公共子序列

（1）最优子结构性质

设序列$X=\{x_1,x_2,\ldots,x_m\}$和$Y=\{y_1,y_2,\ldots,y_n\}$的最长公共子序列为$Z=\{z_1,z_2,\ldots,z_k\}$，则

(a)若$x_m= y_n$，则$z_k=x_m=y_n$，且z_{k-1}是x_{m-1}和y_{n-1}的最长公共子序列。

(b)若$x_m\neq y_n$且$z_k\neq x_m$，则Z是x_{m-1}和Y的最长公共子序列。

(c)若$x_m\neq y_n$且$z_k\neq y_n$，则Z是X和y_{n-1}的最长公共子序列。

由此可见，2个序列的最长公共子序列包含了这2个序列的前缀的最长公共子序列。因此，最长公共子序列问题具有**最优子结构性质**。

典型案例

- 最长公共子序列

（2）建立递归关系

用c[i][j]记录序列X_i和Y_j的最长公共子序列的长度。其中，$X_i=\{x_1,x_2,\ldots,x_i\}$；$Y_j=\{y_1,y_2,\ldots,y_j\}$。当i=0或j=0时，空序列是$X_i$和$Y_j$的最长公共子序列。故此时C[i][j]=0。其它情况下，由最优子结构性质可建立递归关系如下：

$$c[i][j]=\begin{cases}0 & i=0, j=0\\ c[i-1][j-1]+1 & i,j>0; x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0; x_i\neq y_j\end{cases}$$

推导

典型案例

实践

- 最长公共子序列

（3）计算最优值

子问题空间总共有θ(mn)个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

```
void LCSLength(int m，int n，char *x，char *y，int **c，int **b) {
  int i，j;
  for (i = 1; i <= m; i++)  c[i][0] = 0;
  for (i = 1; i <= n; i++)  c[0][i] = 0;
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
      if (x[i]==y[j]) {
        c[i][j]=c[i-1][j-1]+1; b[i][j]=1;}
      else if (c[i-1][j]>=c[i][j-1]) {
          c[i][j]=c[i-1][j]; b[i][j]=2;}
      else { c[i][j]=c[i][j-1]; b[i][j]=3; }
    }
}
```

C[i][j]存储X_i和Y_j的最长公共子序列的长度，b[i][j]记录C[i][j]的值是由哪一个子问题的解得到的，后面构造最长公共子序列时需要用到。

算法时间复杂度为O(mn)

典型案例

- 最长公共子序列

（4）构造最优解

从b[m][n]开始，在数组b中搜索，当b[i][j]=1时，表示X_i和Y_j的最长公共子序列是由X_{i-1}和Y_{j-1}的最长公共子序列在尾部加上x_i所得到的子序列；当b[i][j]=2时，表示表示X_i和Y_j的最长公共子序列与X_{i-1}和Y_j的最长子序列相同；当b[i][j]=3时，表示表示X_i和Y_j的最长公共子序列与X_i和Y_{j-1}的最长子序列相同。

```
void LCS(int i，int j，char *x，int **b) {
   if (i ==0 || j==0) return;
   if (b[i][j]== 1){ LCS(i-1，j-1，x，b); cout<<x[i]; }
   else if (b[i][j]== 2) LCS(i-1，j，x，b);
   else LCS(i，j-1，x，b);
}
```

算法时间复杂度为O(m+n)

典型案例

例

- 最长公共子序列

		B	D	C	A	B	A
	c[m][n]	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

典型案例

- 最长公共子序列

例

X={A, B, C, B, D, A, B}, Y={B, D, C, A, B, A}

b[m][n]	1B	2D	3C	4A	5B	6A
1A	2	2	2	1	3	1
2B	1	3	3	2	1	3
3C	2	2	1	3	2	2
4B	2	2	2	2	1	3
5D	2	2	2	2	2	2
6A	2	2	2	1	2	1
7B	2	2	2	2	2	2

1 ↖

2 ↑

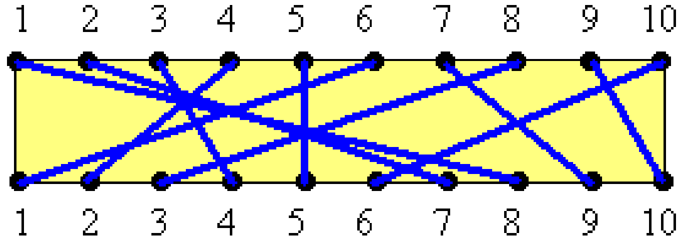
3 ←

典型案例

- **电路布线**

在一块电路板的上、下两端分别有n个接线柱。根据电路设计，要求用导线$(i,\pi(i))$将上端接线柱与下端接线柱相连，其中$\pi(i)$是$\{1,2,\ldots,n\}$的一个排列。导线$(i,\pi(i))$称为该电路板上的第i条连线。对于任何$1\le i<j\le n$，第i条连线和第j条连线相交的充分且必要的条件是$\pi(i)>\pi(j)$。

电路布线问题要确定将哪些连线安排在第一层上，使得该层上有尽可能多的连线。换句话说，该问题要求确定导线集$Nets=\{(i,\pi(i)),1\le i\le n\}$的最大不相交子集。



典型案例

分析讨论

- 电路布线问题—最优子结构性质

记 $N(i,j)=\{t\mid(t,\pi(t))\in Nets, t\le i, \pi(t)\le j\}$， N(i,j)的最大不相交子集为MNS(i,j)。 Size(i,j)=|MNS(i,j)|。

(1)当i=1时

$$Size(1,j)=\begin{cases}0 & j<\pi(1)\\ 1 & j\ge\pi(1)\end{cases}$$

(2)当i>1时

$$Size(i,j)=\begin{cases}Size(i-1,j) & j<\pi(i)\\ \max\{Size(i-1,j), Size(i-1,\pi(i)-1)+1\} & j\ge\pi(i)\end{cases}$$

与最长公共子序列问题同构

典型案例

实践

- 电路布线问题—递归计算最优值

```
void MNS (int C[], int n, int** size) {
    for (int j = 0; j < C[1]; j++) size[1][j] = 0;
    for (int j = C[1]; j <= n; j++) size[1][j] = 1;
    for (int i = 2; i < n; i++) {
        for (int j = 0; j < C[i]; j++) size[i][j] = size[i-1][j];
        for (int j = C[i]; j <= n; j++)
            size[i][j] = max(size[i-1][j], size[i-1][C[i]-1]+1);
    }
    size[n][n] = max(size[n-1][n], size[n-1][C[n]-1]+1);
}
```

时间复杂度$O(n^2)$

典型案例

- 电路布线问题—构造最优解

```
void Traceback (int C[],  int** size, int n, int Net[], int& m) {
    int j = n;
    m = 0;
    for (int i = n; i > 1; i--)
        if (size[i][j] != size[i-1][j]) {
            Net[m++] = i; j = C[i] – 1;
        }
    if (j>= C[1]) Net[m++] = 1;
}
```

时间复杂度O(n)

典型案例

分析理解

- 图像压缩问题

图象的变位压缩存储格式将所给的象素点序列$\{p_1,p_2,\ldots,p_n\}$，$0\le p_i\le 255$分割成m个连续段$S_1,S_2,\ldots,S_m$。第i个象素段S_i中($1\le i\le m$)，有l[i]个象素,且该段中每个象素都只用b[i]位表示。设 $t[i]=\sum_{k=1}^{i-1} l[k]$，则第i个象素段$S_i$为 $S_i=\{p_{t[i]+1},\cdots,p_{t[i]+l[i]}\}$。

设 $$h_i=\left\lceil \log\left(\max_{t[i]+1\le k\le t[i]+l[i]} p_k+1\right)\right\rceil$$

则$h_i\le b[i]\le 8$。因此需要用3位表示b[i],如果限制$1\le l[i]\le 255$，则需要用8位表示l[i]。因此，第i个象素段所需的存储空间为l[i]*b[i]+11位。按此格式存储象素序列$\{p_1,p_2,\ldots,p_n\}$，需要位存储空间为 $\sum_{i=1}^{m} l[i]*b[i]+11m$

典型案例

例

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

设输入的灰度值为P={10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1},

划分1: $S_1=\{10, 12, 15\}, S_2=\{255\}, S_3=\{1, 2, 1, 1, 2, 2, 1, 1\}$

划分2: $S_1=\{10, 12, 15, 255, 1, 2, 1, 1, 2, 2, 1, 1\}$

划分3: $S_1=\{10\}, S_2=\{12\}, S_3=\{15\}, S_4=\{255\}, S_5=\{1\}, S_6=\{2\},$
$S_7=\{1\}, S_8=\{1\}, S_9=\{2\}, S_{10}=\{2\}, S_{11}=\{1\}, S_{12}=\{1\}$

不同划分占用位数:

段头11位

划分1: $4 \times 3 + 8 \times 1 + 2 \times 8 + 11 \times 3 = 69$

划分2: $8 \times 12 + 11 \times 1 = 107$

划分3: $4 + 4 + 4 + 8 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 11 \times 12 =$
163

不同划分压缩效果不同，如何寻找最优划分结果

典型案例

分析

- 图像压缩问题—最优子结构性质

设l[i], b[i]，$1 \le i \le m$是$\{p_1, p_2, \ldots, p_n\}$的一个最优分段。l[1], b[1]则是$\{p_1,\ldots,p_{l[1]}\}$的一个最优分段，且l[i]，b[i]，$2 \le i \le m$，是$\{p_{l[1]+1}, \ldots, p_n\}$的一个最优分段。即图象压缩问题满足最优子结构性质。

典型案例

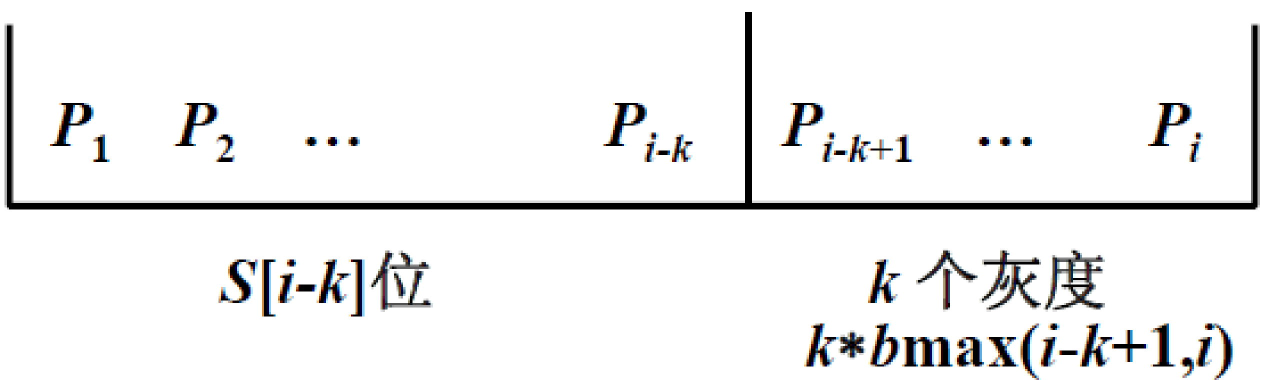
分析理解

- 图像压缩问题—递归计算最优值

设s[i]，1≤i≤n，是象素序列$\{p_1,\ldots,p_i\}$的最优分段所需的存储位数。

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

$$\mathrm{bmax}(i,j)=\left\lceil \log\left(\max_{i\le k\le j}\{p_k\}+1\right)\right\rceil$$



典型案例

实践

- 图像压缩问题—程序设计

```
void Compress (int n, int p[], int s[], int l[], int b[]){
    int Lmax = 256, header = 11;
    s[0] = 0;
    for (int i; i<=n; i++) {
        b[i] = length(p[i]);
        int bmax = b[i];
        s[i] = s[i-1] + bmax;
        l[i] = 1;
        for (int j = 2; j <= i && j <= Lmax; j++){
            if (bmax < b[i-j+1]) bmax = b[i-j+1];
            if (s[i] > s[i-j] + j*bmax) {
                s[i] = s[i-j] + j*bmax;
                l[i] = j;
            }
        }
        s[i] += header;
    }
}
```

```
int length (int i){
    int k = 1; i = i/2;
    while (i > 0) {
        k++;
        i =i/2;
    }
    return k;
}
```

由于算法Compress中对k的循环次数不超这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)。

典型案例

- 图像压缩问题—程序设计

```
void Traceback (int n, int& I, int s[], int l[]) {
    if (n == 0) return;
    Traceback (n – l[n], i, s, l);
    s[i++] = n – l[n];
}
```

```
void Output (int s[], int l[], int b[], int n) {
    cout << “The optimal value is ” << s[n] << endl;
    int m = 0;
    Traceback (n, m, s, l);
    s[m] = n;
    cout << “Decompose into ” << m << “segments” << endl;
    for (int j = 1; j <= m; j++) {
        l[j] = l[s[j]]; b[j] =b[s[j]];
    }
    for (int j = 1; j<=m; j++) {
        cout << l[j] << ‘ ’ <<b[j] <<endl;
    }
}
```

典型案例

例

- 图像压缩问题—算法过程

{10, 12, 15}, {255}, {1, 2}

P = {10, 12, 15, 255, 1, 2}

s[0] = 0
s[1] = s[0] + 1*bmax(p[1]) + header = 0 + 4 + 11 = 15;   l[1] = 1;
s[2] = min{s[1] + 1*bmax(p[2]), s[0] + 2*bmax(p[1], p[2])} + header = 19; l[2] = 2;
s[3] = min{s[2] + 1*bmax(p[3]), s[1] + 2*bmax(p[2],p[3]), s[0] + 3*bmax(p[1],p[2],p[3])} + header = 23;   l[3] = 3;
s[4] = min{s[3] + 1*bmax(p[4]), s[2] + 2*bmax(p[3],p[4]), s[1] + 3*bmax(p[2],p[3],p[4]), s[0] + 4*bmax(p[1],p[2],p[3],p[4])} + header = 42 ;   l[4] = 1;
s[5] = min{s[4] + 1*bmax(p[5]), s[3] + 2*bmax(p[4],p[5]), s[2] + 3*bmax(p[3],p[4],p[5]), s[1] + 4*bmax(p[2],p[3],p[4],p[5]), s[0] + 5*bmax(p[1],p[2],p[3],p[4],p[5])} + header = 50;   l[5] = 2;
s[6] = min{s[5] + 1*bmax(p[6]), s[4] + 2*bmax(p[5],p[6]), s[3] + 3*bmax(p[4],p[5],p[6]), s[2] + 4*bmax(p[3],p[4],p[5],p[6]), s[1] + 5*bmax(p[2],p[3],p[4],p[5],p[6]), s[0] + 6*bmax(p[1],p[2],p[3],p[4],p[5],p[6]), } + header = 57 ;   l[6] = 2;

典型案例

分析理解

- 最大子段和问题

问题：给定n个整数（可以为负数）的序列（$a_1, a_2, \ldots, a_n$），求其最大子段和 $\max\{0, \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k\}$

实例：给定序列A = (-2, 11, -4, 13, -5, -2)
长度为1的子段和有： -2, 11, -4, 13, -5, -2
长度为2的子段和有： 9, 7, 9, 8, -7
长度为3的子段和有： 5, 20, 4, 6
长度为4的子段和有： 18, 15, 2
长度为5的子段和有： 13, 13
长度为6的子段和有： 11
最大子段和为11 – 4 + 13 = 20

典型案例

分析

- 最大子段和问题—蛮力法

```
int MaxSum (int n, int* a, int& besti, int& bestj) {
    int sum = 0MIN;
    for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++){
        int thissum = 0;
        for (k = i; k <= j; k++) thissum += a[k];
        if (thissum > sum) {
            sum = thissum;
            besti = i;
            bestj = j;
        }
    }
    return sum;
}
```

时间复杂度$O(n^3)$

典型案例

分析

- 最大子段和问题—蛮力法改进

$$\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$$

时间复杂度$O(n^2)$

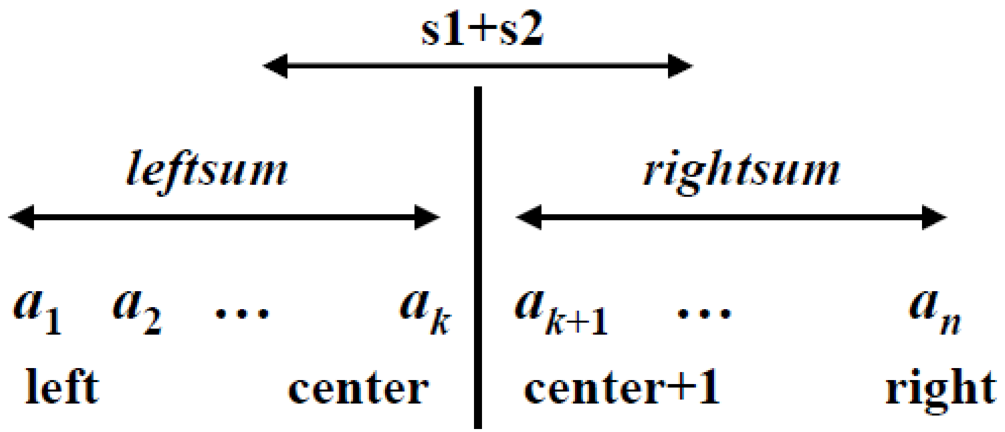
```
int MaxSum (int n, int* a, int& besti, int& bestj) {
    int sum = 0MIN;
    for (int i = 1; i <= n; i++) {
        int thissum = 0;
        for (int j = i; j <= n; j++){
            thissum += a[j];
            if (thissum > sum) {
                sum = thissum;
                besti = i;
                bestj = j;
            }
        }
    }
    return sum;
}
```

典型案例

分析

- 最大子段和问题—分治法

将序列分为左右两半，中间分点为center;
递归计算左段最大子段和leftsum;
递归计算右段最大子段和rightsum;
$a_{center} \rightarrow a_1$的最大和s1， $a_{center} \rightarrow a_n$的最大和s2;
问题解为max{leftsum, rightsum, s1+s2}。



典型案例

- 最大子段和问题—分

$$T(n)=\begin{cases} O(1) & n\le c \\ 2T(n/2)+O(n) & n>c \end{cases}$$

O(nlogn)

```
int MaxSum (int n, int* a) {
   return MaxSubSum (a, 1, n);
}
```

```
int MaxSubSum (int* a, int left, int right) {
   int sum = 0;
   if (left == right) sum = a[left] > 0 ? a[left]:0;
   else{
      int center = (left + right)/2;
      int leftsum = MaxSubSum (a, left, center);
      int rightsum = MaxSubSum (a, center + 1, right);
      int s1 = 0; int lefts = 0;
      for (int i = center; i >= left; i--) {
         lefts += a[i];
         if (lefts > s1) s1 = lefts;
      }
      int s2 = 0; int rights = 0;
      for (int i = center + 1; i <= right; i++) {
         rights += a[i];
         if (rights > s2) s2 = rights;
      }
      sum = s1 + s2;
      if (sum < leftsum) sum = leftsum;
      if (sum < rightsum) sum = rightsum;
   }
   return sum;
}
```

典型案例

分析讨论

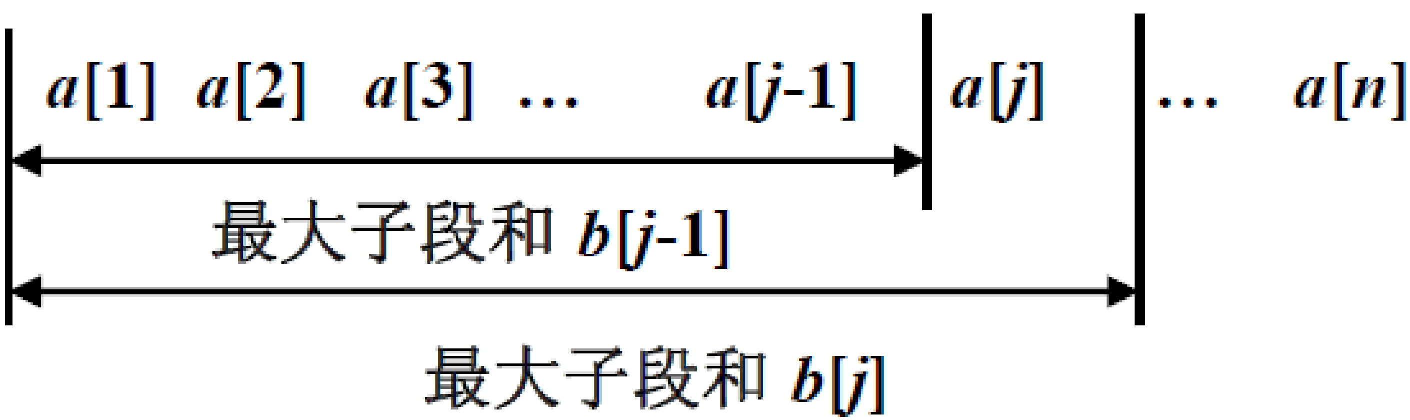
- 最大子段和问题—动态规划

设b[j]表示最后一项为a[j]的序列构成的最大的子段和最优解b[1],b[2],…,b[n]中的最大值，即

$$b[j]=\max_{1\le i\le j}\{\sum_{k=i}^{j}a[k]\},1\le j\le n$$

可建立b[j]的递推方程

$$b[j]=\max\{b[j-1]+a[j],a[j]\}$$



典型案例

实践

- 最大子段和问题

```
int MaxSum (int n, int* a) {
    int sum = MIN, b = 0;
    for (i = 1; i <= n; i++) {
        if (b > 0) b += a[i];
        else b = a[i];
        if (b > sum) sum = b;
    }
    return sum;
}
```

实例：给定序列A = (2, -5, 8, 11, -3, 4, 6)

i=1: b=a[i]=2, sum=b=2;

i=2: b=2+a[2]=-3, sum=2;

i=3: b=a[3]=8, sum=8;

i=4: b=8+a[4]=19, sum=19;

i=5: b=19+a[5]=16, sum=19;

i=6: b=16+a[6]=20, sum=20;

i=7: b=20+a[7]=26, sum=26.

时间复杂度O(n)

典型案例

分析理解

- 投资问题

问题：m元钱，n个投资项目，函数$f_i(x)$表示将x元钱投入第i个项目所产生的效益，i=1,2,…n。

问题：如何分配这m元钱，使得投资的总效益最高？

目标函数：$\max\{f_1(x_1)+f_2(x_2)+\ldots+f_n(x_n)\}$

约束条件：$x_1+x_2+\ldots+x_n=m,\ \ x_i\in N$

典型案例

- 投资问题—实例分析

有5万元钱，4个项目，效益函数（以万元为单位）如表所示。

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

典型案例

分析讨论

- 投资问题—最优子结构分析

设$F_k(x)$表示x万元投给前k个项目的最大收益，其中k=1,2,…n，x=1,2,…m。可以得到投资收益函数递归关系：

$$F_k(x)=\begin{cases} f_1(x) & k=1 \\ \max\limits_{0\le x_k\le x}\{f_k(x_k)+F_{k-1}(x-x_k)\} & k>1 \end{cases}$$

典型案例

例

- 投资问题—计算过程

X	$F_1(x)$ $x_1(x)$	$F_2(x)$ $x_2(x)$	$F_3(x)$ $x_3(x)$	$F_4(x)$ $x_4(x)$
1				
2				
3				
4				
5				

最优解：$x_1=$, $x_2=$, $x_3=$, $x_4=$;
$F_4(5)=$.

典型案例

例

- 投资问题—计算过程

X	$F_1(x)$	$x_1(x)$	$F_2(x)$	$x_2(x)$	$F_3(x)$	$x_3(x)$	$F_4(x)$	$x_4(x)$
1	11	1	11	0	11	0	20	1
2	12	2	12	0	13	1	31	1
3	13	3	16	2	30	3	33	1
4	14	4	21	3	41	3	50	1
5	15	5	26	4	43	4	61	1

最优解：$x_1=1, x_2=0, x_3=3, x_4=1;$
$F_4(5)=61.$

典型案例

- 投资问题—算法复杂性分析

设n个项目，钱数为m.
除k=1外，对于项$F_k(x)$ $(2 \le k \le n,\ 1 \le x \le m)$的计算需要x+1次加法和x次比较。对k求和，算法执行加法次数满足:

$$\sum_{k=2}^{n}\sum_{x=1}^{m}(x+1)=\frac{1}{2}(n-1)m(m+3)$$

比较次数满足:

$$\sum_{k=2}^{n}\sum_{x=1}^{m}x=\frac{1}{2}(n-1)m(m+1)$$

时间复杂度$O(nm^2)$

$$F_k(x)=\begin{cases} f_1(x) & k=1 \\ \max\limits_{0\le x_k\le x}\{f_k(x_k)+F_{k-1}(x-x_k)\} & k>1 \end{cases}$$

典型案例

分析理解

- 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。

问题：应如何选择装入背包中的物品，使得装入背包中物品的总价值最大。

说明：在选择装入背包的物品时，对每种物品i只有两个选择，装入背包或不装入背包，也不能将物品装入背包多次。

给定$c>0$，$w_i>0$，$v_i>0$，$1\le i\le n$，要求找出一个n元0-1向量$(x_1,x_2,\ldots x_n)$，$x_i\in\{0,1\}$，使得

$\sum_{i=1}^{n} w_i x_i \le c$ 且 $\sum_{i=1}^{n} v_i x_i$ 达到最大。

典型案例

分析讨论

- 0-1背包问题—递推方程

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

设子问题最优值为m(i，j)，即m(i，j)是背包容量为j，可选择物品为1, 2, … , i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:

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

$$\begin{cases} m(0, j) = 0 \\ m(i,0) = 0 \end{cases}$$

n=5, c=10,
w={2, 2, 6, 5, 4}
v={6, 3, 5, 4, 6}

典型案例

例

- 0-1背包问题—实例求解分析

k \ c		1	2	3	4	5	6	7	8	9	10
	$F_k(y)$	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	14	14
4	0	0	6	6	9	9	9	10	11	14	14
5	0	0	6	6	9	9	12	12	15	15	15

典型案例

- 0-1背包问题—程序设计—求解最优值|构造最优解

时间复杂度:
Knapsack: $O(nc)$
Traceback: $O(n)$

伪多项式时间，当$c>2^n$时，算法Knapsack需要$\Omega(n2^n)$

典型案例

- 0-1背包问题—另外一种递推关系

0-1背包问题的子问题结构为

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

设子问题最优值为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：

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

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

n=5, c=10,
w={2, 2, 6, 5, 4}
v={6, 3, 5, 4, 6}

典型案例

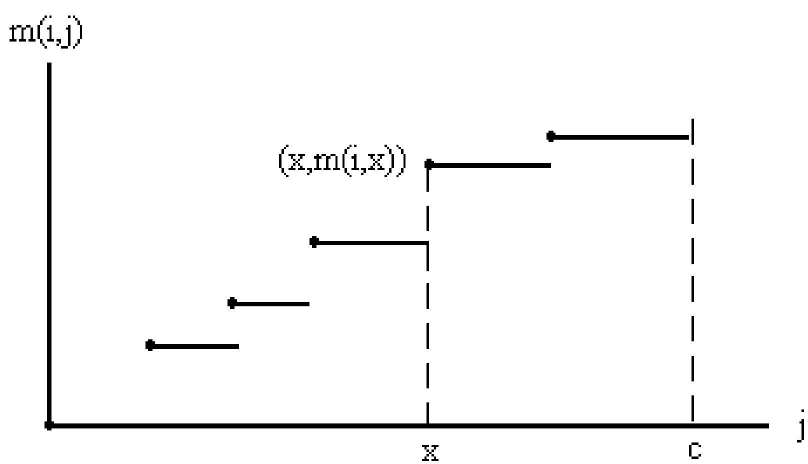
- 0-1背包问题—实例求解分析

M[i][j]	1	2	3	4	5	6	7	8	9	10
0	0	6	6	9	9	12	12	15	15	15
0	0	3	3	6	6	9	9	+V[1] 9	10	11
0	0	0	0	6	6	6	6	6	10	11
0	0	0	0	+V[3] 6	6	6	6	6	10	10
+V[4] 0	0	0	0	+V[4] 6	6	6	6	6	6	6

典型案例

- 0-1背包问题—算法优化改进

根据m(i,j)的递归式，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。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)，可用一个表p[i]存储函数m(i，j)的全部跳跃点。表p[i]可依计算m(i，j)的递归式递归地由表p[i+1]计算，初始时p[n+1]={(0，0)}。

典型案例

分析理解

- 0-N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。
问题：应如何选择装入背包中的物品，使得装入背包中物品的总价值最大。
说明：**同一物品可以装入背包多次。**

目标函数

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

约束条件

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le c \\ x_i \in N \end{cases}$$

典型案例

分析讨论

- 0-N背包问题—递推关系

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

$$\begin{cases}m(0,j)=0\\ m(i,0)=0\end{cases}$$

典型案例

- 0-N背包问题—程序设计—求解最优值|构造最优解

n=4, c=10
w={2, 3, 4, 7}
v={1, 3, 5, 9}

典型案例

例

- 0-N背包问题——实例求解分析

$F_k(y)$

k \ y	1	2	3	4	5	6	7	8	8	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	**12**

$-\infty+v_k$

$F_k(y-3)+v_k$

$F_k(y-3)+v_k$

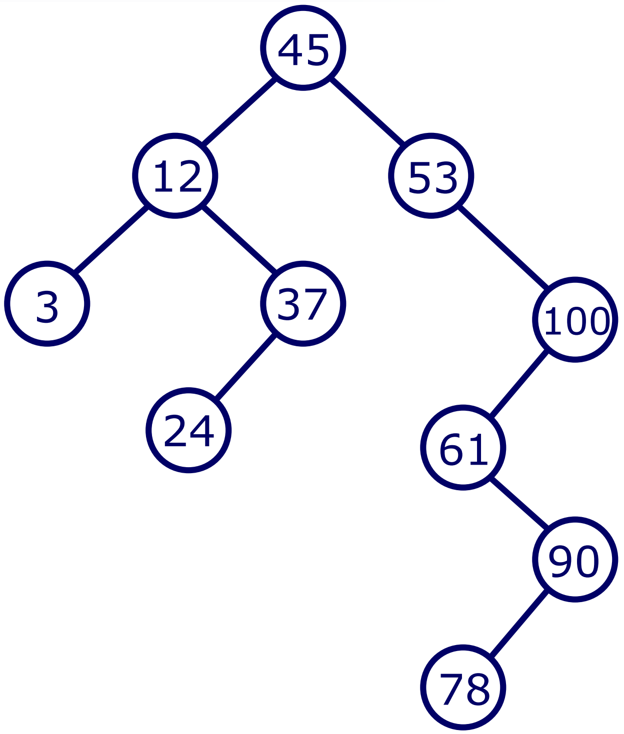
$F_k(y-4)+v_k$

$F_k(y-7)+v_k$

典型案例

- 最优二叉搜索树

二叉排序树:
(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排序树。



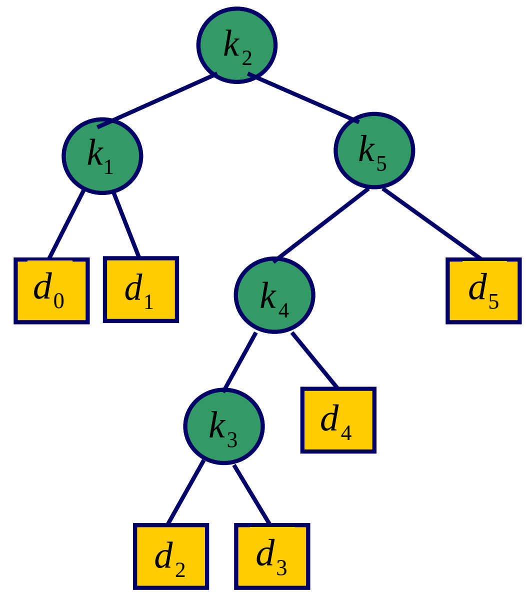
随机的情况下，二叉查找树的平均查找长度和logn是等数量级的。

典型案例

- 最优二叉搜索树

设树的根节点深度为0

二叉搜索树的叶结点是形如(x_i, x_{i+1})的开区间。在二叉搜索树中搜索一个元素x，返回结果有两种情形：
(1)在树的内结点找到$x=x_i$;
(2)在树的叶结点中确定$x\in(x_i, x_{i+1})$。



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

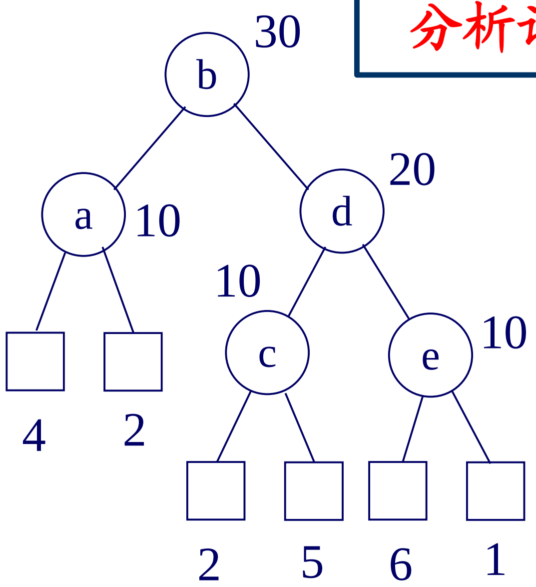
$$E(\text{search cost in } T) = \sum_{i=1}^{n} (\text{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i)\cdot q_i$$

典型案例

- 最优二叉搜索树—实例分析

分析讨论

设数据集S={a, b, c, d, e}，存储S的二叉搜索树如图所示。
设被查找的x值对应树中各结点的分布概率为
P=(4, 10, 2, 30, 2, 10, 5, 20, 6, 10, 1)/100



平均查找长度:

$$L=0.3\times 1+(0.1+0.2)\times 2+(0.1+0.1)\times 3$$
$$+(0.04+0.02)\times 2+(0.02+0.05+0.06+0.01)\times 3$$
$$=2.04$$

这棵树构造的是否最好?

典型案例

- ## 最优二叉搜索树—问题描述

给定数据集合S与存取概率分布P，求一颗平均查找次数（平均查找长度）最小的二叉搜索树，即最优二叉搜索树。

蛮力法:
枚举具有n个结点的所有二分搜索树，计算平均查找长度，从中找出最优树。
复杂度与矩阵链相乘问题相同，为Catalan数，下界为$\Omega(4^n/n^{3/2})$

典型案例

- 最优二叉搜索树—最优子结构性质

问题满足优化原则

设s[i][j]=<x_i, x_{i+1}, …, x_j>是s以i和j作为边界的子数据集，
p[i][j]=<a_{i-1}, b_i, a_i, b_{i+1}, …, b_j, a_j>是对应s[i][j]的存取概率分布。

例： s=<a, b, c, d, e>， p=<4, 10, 2, 30, 2, 10, 5, 20, 6, 10, 1>/100
s[2][4]=<b, c, d>， p[2][4]=< 2, 30, 2, 10, 5, 20, 6 >/100

设以x_k为根，可将原问题划分为两个子问题：
s[i][k-1], p[i][k-1]
s[k+1][j], p[k+1][j]

例如以b为根，可将s，p划分为
s[1][1]=<a>， p[1][1]=<4, 10, 2>/100
s[3][5]=<c, d, e>， p[3][5]=< 2, 10, 5, 20, 6, 10, 1>/100

典型案例

分析讨论

- 最优二叉搜索树—递推关系

设m[i][j]是相对于输入s[i][j]和p[i][j]的最优二叉搜索树的平均比较次数，令p[i][j]中所有概率（包括数据元素与空隙）之和为w[i,j]，则

$$w[i,j]=\sum_{p=i-1}^{j}a_p+\sum_{q=i}^{j}b_q$$

$$m[i,j]=\min_{i\le k\le j}\{m[i,k-1]+m[k+1][j]+w[i][j]\},1\le i\le j\le n$$

$$m[i,i-1]=0,\quad i=1,2,\ldots,n$$

与矩阵链乘积问题、凸多边形最优三角剖分问题同构。

典型案例

- 最优二叉搜索树—递推关系

令m[i][j]$_k$表示根为x_k时的二分搜索树平均比较次数最小值，则

$$
\begin{aligned}
m[i,j]_k &= (m[i,k-1]+w[i,k-1])+(m[k+1,j]+w[k+1,j])+1\times b_k \\
&= (m[i,k-1]+m[k+1,j])+(w[i,k-1]+b_k+w[k+1,j]) \\
&= (m[i,k-1]+m[k+1,j])+(\sum_{p=i-1}^{k-1} a_p+\sum_{q=i}^{k-1} b_q)+b_k+(\sum_{p=k}^{j} a_p+\sum_{q=k+1}^{j} b_q) \\
&= (m[i,k-1]+m[k+1,j])+\sum_{p=i-1}^{j} a_p+\sum_{q=i}^{j} b_q \\
&= m[i,k-1]+m[k+1,j]+w[i,j]
\end{aligned}
$$

典型案例

- 最优二叉搜索树—程序设计

```
void OptimalBinarySearchTree (int a, int b, int n, int** m, int** s, int** w){
    for (int i = 0; i <= n; i++) {w[i+1][i] = a[i]; m[i+1][i] = 0;}
    for (int r = 0; r < n; r++)
        for (int i = 1; i <= n-r; i++) {
            int j = i + r;
            w[i][j] = w[i][j-1] + a[j] +b[j];
            m[i][j] = m[i+1][j];
            s[i][j] = i;
            for (int k = i+1; k <= j; k++){
                int t = m[i][k-1] + m[k+1][j];
                if (t < m[i][j]) {m[i][j] = t; s[i][j] = k;}
            }
        m[i][j] += w[i][j];
        }
}
```

时间复杂度$O(n^3)$
空间复杂的 $O(n^2)$

s[i][j]保存最优子树T(i, j)的根节点中元素，由s[i][j]可在O(n)时间内构造所求的最优二叉搜索树。

s=<A, B, C, D, E>,
p=<0.04, 0.1, 0.02, 0.3, 0.02, 0.1, 0.05, 0.2, 0.06, 0.1, 0.01>

典型案例

- 最优二叉搜索树—实例求解分析

r=0: m[1][1]=0.16, m[2][2]=0.34, m[3][3]=0.17, m[4][4]=0.31, m[5][5]=0.17;
r=1:
m[1][2]=min{m[2][2], m[1][1]}+0.48=0.64, s[1][2]=2;
m[2][3]=min{m[3][3], m[2][2]}+0.49=0.66, s[2][3]=2;
m[3][4]=min{m[4][4], m[3][3]}+0.43=0.60, s[3][4]=4;
m[4][5]=min{m[5][5], m[4][4]}+0.42=0.59, s[4][5]=4;
r=2:
m[1][3]=min{m[2][3], m[1][1]+m[3][3], m[1][2]}+0.63=0.96, s[1][3]=2;
m[2][4]=min{m[3][4], m[2][2]+m[4][4], m[2][3]}+0.75=1.35, s[2][4]=2;
m[3][5]=min{m[4][5], m[3][3]+m[5][5], m[3][4]}+0.54=0.88, s[2][4]=4;
r=3:
m[1][4]=min{m[2][4], m[1][1]+m[3][4], m[1][2]+m[4][4], m[1][3]}+0.89=1.65, s[1][4]=2;
m[2][5]=min{m[3][5], m[2][2]+m[4][5], m[2][3]+m[5][5], m[2][4]}+0.86=1.69, s[1][4]=4;
r=4:
m[1][5]=min{m[2][5], m[1][1]+m[3][5], m[1][2]+m[4][5], m[1][3]+m[5][5], m[1][4]}+1
=2.04, s[1][5]=2.

典型案例

分析理解

- 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。

问题：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。

典型案例

分析讨论

- 流水作业调度—问题分析

（1）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。
（2）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M_1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)。

典型案例

- 流水作业调度—最优子结构性质

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M2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M2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。

典型案例

- 流水作业调度—递推关系

由流水作业调度问题的最优子结构性质分析可知:

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

推广到一般情况:

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

典型案例

- 流水作业调度—Johnson不等式

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**。

典型案例

- 流水作业调度—Johnson不等式

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

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

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

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

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

从而，$T_{ij} \le T_{ji}$， $T(S,t) \le T'(S,t)$

典型案例

- 流水作业调度—Johnson不等式

当作业i和作业j不满足Johnson不等式时，交换它们的加工顺序后，不增加加工时间。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

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

所有满足Johnson法则的调度均为最优调度。

典型案例

- 流水作业调度—Johnson算法

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

典型案例

- 流水作业调度—Johnson算法

```
int FlowShop (int n, int a, int b, int c) {
    class Jobtype {
        public:
            int operator <= (Jobtype a) const{
                return (key <= a.key);
            }
            int key, index;
            bool job;
    };
    Jobtype* d = new Jobtype[n];
    for (int i = 0; i < n; i++) {
        d[i].key = a[i]>b[i] ? b[i] : a[i];
        d[i].job = a[i] <= b[i];
        d[i].index = i;
    }
```

```
    sort(d, n);
    int j = 0, k = n-1;
    for (int i=0; i<n; i++){
        if (d[i].job) c[j++] = d[i].index;
        else c[k--] = d[i].index;
    }
    j = a[c[0]];
    k = j + b[c[0]];
    for (int i=1; i<n; i++) {
        j += a[c[i]];
        k = j < k ? k+b[c[i]] : c[c[i]];
    }
    delete d;
    return k;
}
```

典型案例

分析理解

序列匹配—问题描述

设$S_1[1,m]$和$S_2[1,n]$为两个字符序列，对这两个字符序列进行顺序扫描比较，比较时可以在某个序列的两个字符之间插入空格，以使他们局部相同区域对应。在比较中，如果两个对应字符相等，赋权值2，如果对应字符不等，赋权值-2，如果一个是字符一个是空格，赋权值-1,。
问题：寻找权值最大的匹配度（匹配权值和）与对准方式。

例：序列axabcdes和axbacfes的两种对准方式如下：

a x a b - c d e s a x - b a c f e s	a x - a b c d e s a x b a - c f e s

这两种对准方式的匹配度均为：$6 \times 2 + (-2) + 2 \times (-1) = 8$

典型案例

分析讨论

- 序列匹配—问题分析

设c[i,j]表示序列$s_1[1,i]$与$s_2[1,j]$对比的最大权值。

(1) $s_1[i]$与$s_2[j]$对准情况:

a) 若$s_1[i] = s_2[j]$，问题规约为$s_1[1,i-1]$ 与$s_2[1,j-1]$的对准，权值为c[i-1,j-1]+2;

b) 若$s_1[i] \neq s_2[j]$，问题仍规约为$s_1[1,i-1]$ 与$s_2[1,j-1]$的对准，权值为c[i-1,j-1]-2;

(2) $s_1[i]$与$s_2[j]$不对准情况:

需要在序列$s_1[1,i]$或$s_2[1,j]$后增加空格，问题归结为$s_1[1,i-1]$与$s_2[1,j]$的对准，或者$s_1[1,i]$ 与$s_2[1,j-1]$的对准，权值为子问题权值-1.

典型案例

- 序列匹配—递推方程

$$c[i][j] = \max_{\substack{0 \le i \le m \\ 0 \le j \le n}} \{0, c[i-1][j-1] + t(s_1[i], s_2[j]), c[i][j-1] - 1, c[i-1][j] - 1\}$$

$$t(s_1[i], s_2[j]) = \begin{cases} 2 & s_1[i] = s_2[j] \\ -2 & s_1[i] \ne s_2[j] \end{cases}$$

$$c[0][j] = 0$$
$$c[i][0] = 0$$

时间复杂度O(mn)

课堂演示

典型案例

- 序列匹配——程序设计

典型案例

- …

开放性讨论

Summary