Chapter 15. 动态规划

Ch.15 动态规划

动态规划主要用于优化问题求解，即求出问题的最优（最大/小）解，当有多个最优解时一般是求一个即可。

- 与分治法异同
  - 相同点：都是通过合并子问题的解来解决整个问题的解
  - 不同点

  1)分治法是将问题划分为独立的子问题，递归地解子问题，然后合并

Ch.15 动态规划

2)当分解子问题，但他们共享子子问题时，可采用动态规划

因为此时分治法将重复地解这些共同的子子问题，形成重复计算，而动态规划对每一子子问题只做一次计算，然后将答案存储在一表中（这就是programming含义，像节目单一样），故可避免重复计算

Ch.15 动态规划

- **四个步骤**
  - Step1：描述最优解的结构特征
  - Step2：递归地定义一个最优解的值
  - Step3：自底向上计算一个最优解的值
  - Step4：从已计算的信息中构造一个最优解

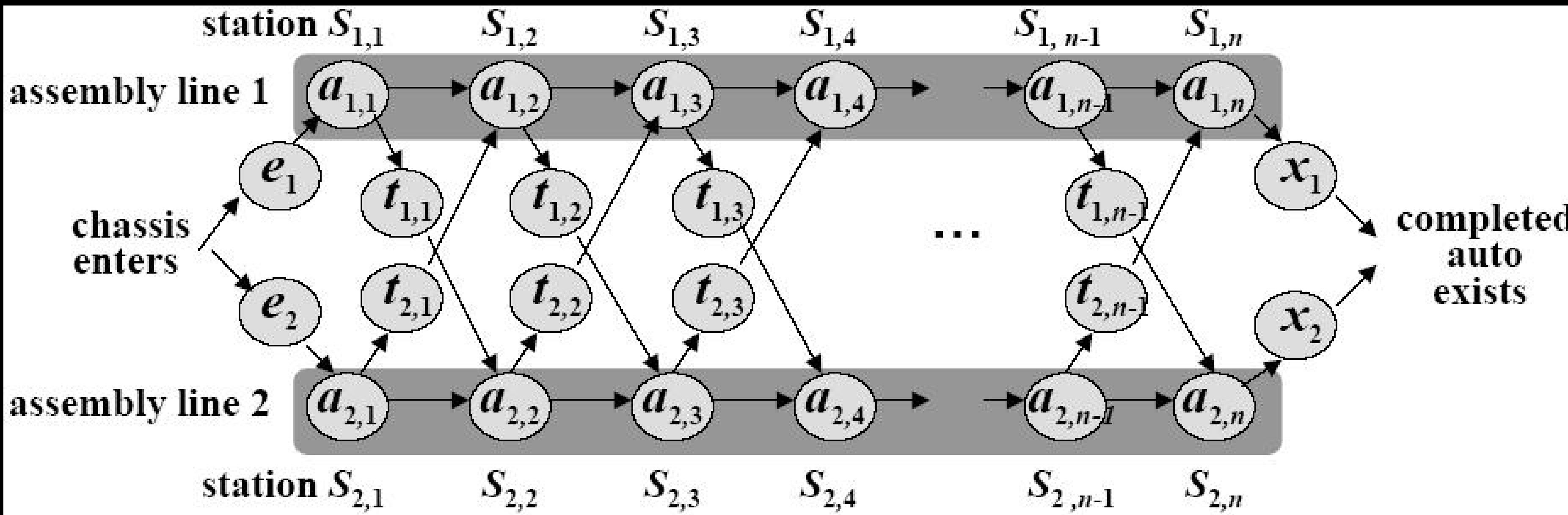
Step1、2、3是基础

若要构造最优解，则step3中应维护附加信息，如求最短路径

装配线调度问题

- 装配线调度
  - 在一工厂里有两条装配线并行生成汽车, lines 1 and 2
  - 每条装配线有 n 个装配站 $S_{i,1}...S_{i,n}$, i =1, 2
  - 设对于每个 j, $S_{1,j}$ 和$S_{2,j}$ 的功能相同, 但装配时间不同，对$S_{i,j}$ 的装配时间为 $a_{i,j}$
  - 汽车装配可以从一条装配线移到另一条装配线，从装配站$S_{i,j}$ 移到到另一条装配线上的时间为$t_{i,j}$, i=1, 2 and j =1, 2, … , n-1
  - 汽车进入装配线的时间为 e_i，汽车离开装配线的时间为 x_i

装配线调度问题



问题：确定在装配线1内选择哪些站，以及在装配线2内选择哪些站，以使汽车装配的总时间最小。

采用强力法计算，需要2^n时间

装配线调度问题

步骤1：通过工厂最快路线的结构

- 最优子结构：一个问题的最优解包含了子问题的一个最优解，我们称这个性质为最优子结构。
- 通过装配站$S_{1,j}$的最快路线是以下二者之一：

  （1）通过装配站$S_{1,j-1}$的最快路线，然后直接通过装配站$S_{1,j}$

  （2）通过装配站$S_{2,j-1}$的最快路线，从装配线2移动到装配线1，然后通过装配站$S_{1,j}$
- 通过装配站$S_{2,j}$的最快路线是以下二者之一：

  （1）通过装配站$S_{2,j-1}$的最快路线，然后直接通过装配站$S_{2,j}$

  （2）通过装配站$S_{1,j-1}$的最快路线，从装配线1移动到装配线2，然后通过装配站$S_{2,j}$
- 总结：寻找通过任一条装配线上的装配站j的最快路线，我们解决它的子问题，即寻找通过两条装配线上的装配站j-1的最快路线。

装配线调度问题

步骤2：一个递归的解：利用子问题的解来递归定义一个最优解的值

- 设$f_i[j]$ 表示一个汽车从起点到装配站$S_{i,j}$装配的最快可能时间。
- 递归公式:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

- 总的时间:

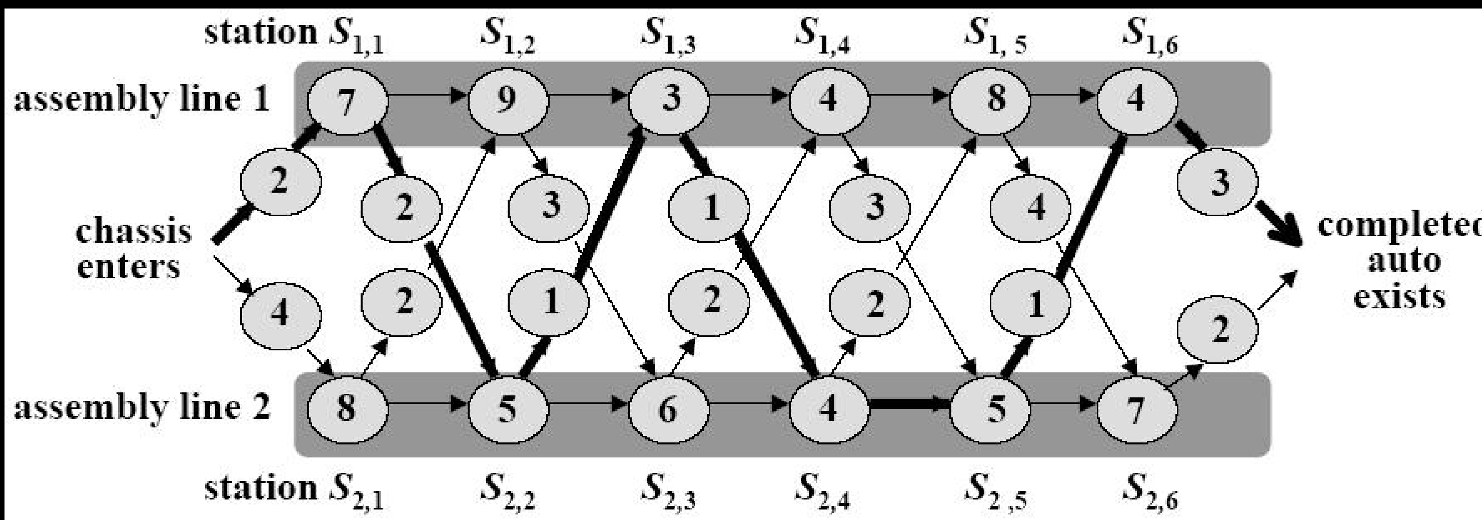
$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

装配线调度问题

- 步骤3：计算最快时间

❖根据递归公式，采用递归算法来计算是一件简单的事情，但是执行时间是关于n的指数形式。

❖采用从左到右（自底向上），迭代的方式进行计算



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

ASSEMBLY-LIN

$l_i[j]$: 为了跟踪最优解的构造过程

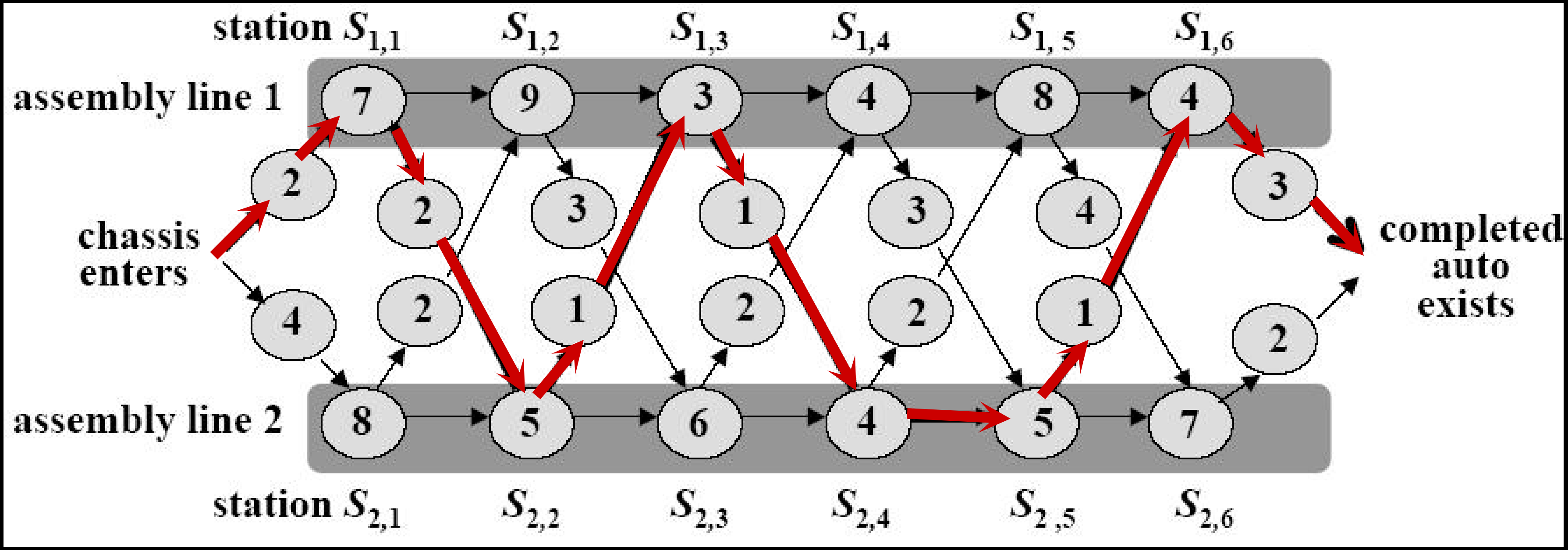
-- $l_i[j]$ 记录了装配站 $S_{i,j}$前一个装配站所在的装配线编号

-- l^* 是第 n 个装配站所在的装配线编号.

```
FASTEST-WAY(a,t,e,x,n)
f_1[1] ← e_1 + a_{1,1}
f_2[1] ← e_2 + a_{2,1}
for j ← 2 to n
  do if f_1[j-1]+a_{1,j} ≤ f_2[j-1]+t_{2,j-1}+a_{1,j}
        then f_1[j] ← f_1[j-1]+a_{1,j}
             l_1[j] ← 1
        else f_1[j] ← f_2[j-1]+t_{2,j-1}+a_{1,j}
             l_1[j] ← 2
     if f_2[j-1]+a_{2,j} ≤ f_1[j-1] +t_{1,j-1}+a_{2,j}
        then f_2[j] ← f_2[j-1]+a_{2,j}
             l_2[j] ← 2
        else f_2[j] ← f_1[j-1]+t_{1,j-1}+a_{2,j}
             l_2[j] ← 1
if f_1[n] + x_1 ≤ f_2[n] + x_2
   then f* = f_1[n] + x_1
        l* = 1
   else f* = f_2[n] + x_2
        l* = 2
```

$\Theta(n)$

Construct an optimal solution



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

Constructing the fastest way

PRINT STATION

```
PRINT-STATIONS(l,n)
i ← l*
print "line" i ",station" n
for j ← n downto 2
   do i ← l_i[j]
      print "line" i ", station" j-1
```

RECURSIVE PRINT STATION

```
RECURSIVE-PRINT-STATIONS(l,i,j)
if j = 0 then return
RECURSIVE-PRINT-STATIONS(l,l_i[j],j-1)
print "line" i ", station" j

Note: To print out all the stations,
   call RECURSIVE-PRINT-STATIONS(l,l*,n)
```

矩阵链乘

- 给定n个矩阵的序列$< A_1, A_2, .., A_n >$，需要计算其积 $A_1 A_2 \cdots A_n$
- 计算多个矩阵积可用括号来决定计算次序，每一个括号内的矩阵相乘调用标准的矩阵乘法
- 矩阵积的完全括号化
  - 它是单个矩阵
  - 或是两个完全括号化的矩阵积被包括在一个括号里

矩阵链乘

- 矩阵乘法满足结合律，故所有完全括号化产生同样积

例：以下是A1~A8积不同的两种完全括号化方式

$$((A_1(A_2(A_3A_4)))((A_5A_6)(A_7A_8)))$$

$$(((A_1A_2)((A_3A_4)A_5))(A_6(A_7A_8)))$$

矩阵链乘

- 不同的括号化方式产生不同的计算成本

两矩阵相乘 $A_{pq} \cdot B_{qr}$ 的数量乘次数为p*q*r

例：设 A_1, A_2, A_3 的维数分别为 $10\times 100, 100\times 5, 5\times 50$

$$((A_1A_2)A_3): \quad A_1A_2 — 10\times 100\times 5 = 5000$$

$$(A_1A_2)A_3 — 10\times 5\times 50 = 2500$$

$$\text{Total}: 5000 + 2500 = 7500$$

$$(A_1(A_2A_3)): \quad A_2A_3 — 100\times 5\times 50 = 25000$$

$$A_1(A_2A_3) — 10\times 100\times 50 = 50000$$

$$\text{Total}: 50000 + 25000 = 75000$$

矩阵链乘

- 矩阵链乘实质上是一个最优括号化问题
  - 给定$< A_1, A_2, \ldots, A_n >$，$A_i$的维数$p_{i-1} \times p_i (1 \le i \le n)$，在$A_1 A_2 \cdots A_n$的积中插入括号使其完全括号化，且使得数量乘法次数最少
  - 计算括号数目

    P(n)表示n个矩阵序列中可选括号数，将该序列从k和k+1间划分为两子序列，然后独立地将其括号化，用穷举法产生的括号数:

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

矩阵链乘

例：以下是A1~A8积不同的两种完全括号化方式

$$((A_1(A_2(A_3A_4)))((A_5A_6)(A_7A_8)))$$

$$(((A_1A_2)((A_3A_4)A_5))(A_6(A_7A_8)))$$

$$((A_1)(A_2A_3A_4A_5A_6A_7A_8)) \quad ((A_1A_2)(A_3A_4A_5A_6A_7A_8))$$

$$((A_1A_2A_3)(A_4A_5A_6A_7A_8)) \quad ((A_1A_2A_3A_4)(A_5A_6A_7A_8))$$

$$((A_1A_2A_3A_4A_5)(A_6A_7A_8)) \quad ((A_1A_2A_3A_4A_5A_6)(A_7A_8))$$

$$((A_1A_2A_3A_4A_5A_6A_7)(A_8))$$

矩阵链乘

- **Step1：最优的括号化结构(即描述最优解的结构特征)**
  - 将 $A_i A_{i+1} \cdots A_j$ 积简记为$A_{i..j}$ ，其中 $1 \le i \le j \le n$
  - 设 $A_i A_{i+1} \cdots A_j$的最优括号化是在 A_k 和A_{k+1} 之间进行分裂 $(i \le k \le j-1, 要求 i < j)$
  - 对某个k，先计算 $A_{i..k}$ 和 $A_{k+1..j}$ ，然后将这两个积相乘产生积 $A_{i..j}$

矩阵链乘

最优括号化的成本是:

计算$A_{i..k}$ 的成本+计算 $A_{k+1..j}$ 的成本+两个积相乘成本

关键: $A_i A_{i+1} \cdots A_j$的最优括号化亦要求前后缀子链$A_{i..k}$和$A_{k+1..j}$是最优括号化。可用反证法证明，若$A_{i..k}$括号化不是最优，则可找到一个成本更小的方法将其括号化，代入到$A_{i..j}$的最优括号化表示中，得到的计算成本比最优解小，矛盾！

矩阵链乘

- **Step2：递归解（递归地定义一个最优解的值）**

怎样用子问题的最优解递归地定义原问题的最优解（一般是最优解的值）？对矩阵链乘，子问题的最优解的值是：

确定$A_{i..j}$括号化的最小代价（即按最优括号化计算的成本）。

设m[i,j]是计算$A_{i..j}$所需乘法的最小次数（最优解的值），则$A_{1..n}$的最小计算成本是m[1,n]。

1)若i=j，m[i,i]=0，1≤i≤n，链上只有一个A_i，无需乘法

矩阵链乘

2)若i<j，由Step1中的最优解结构可知：

假定最优括号化的分裂点为k（ i≤k≤j-1 ），则：

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

其中 $m[i,k]$：子积 $A_{i..k}$ 的最小代价，

p_{i-1} : A_i 的行，p_k : A_k 的列， p_j : A_j 的列

在不知道k的取值的情况下，可在j-i个值中选取最优者，所以$A_{i..j}$的最小计算成本为：

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

若要构造最优解，定义S[i,j]记录分裂点k

矩阵链乘

- **Step3：计算最优解的值**

若简单地用递归算法计算积$A_{1..n}$的最小成本m[1,n]，则算法时间仍为指数阶。但是，满足 $1 \le i \le j \le n$ 的i和j总共只有

$$\binom{n}{2}+n=\Theta(n^2)\ //\ i<j\text{有}\binom{n}{2}\text{个，}\ i=j\text{有n个}$$

即子问题个数并非指数。递归算法在其递归树的不同分支上要重复计数每个子问题，这是动态规划应用的另一特征（即重叠子问题），故自底向上计算m的值。

- 算法

矩阵链乘

- **算法：按链长j-i+1递增序计算m[i,j]**

输入：

$p = \langle p_0, p_1, \ldots, p_n \rangle$, 其中 $p_{i-1} \times p_i$ 是 A_i 的维数

$m[1..n, 1..n]$ 记 录 成 本

$S[1..n, 1..n]$ 记 录 相 应 分 裂 点 k

```
MatrixChainOrder(p){
    n ← length[p] − 1;
    for i ← 1  to  n  do{
        m[i,i] ← 0;
    }
    for l ← 2  to  n  do{// A_{i..j}链长 l = j − i + 1
        //第一次计算m[i,i+1],第二次计算m[i,i+2]等
        for i ← 1  to  n − l + 1  do{
                //1 ≤ i ≤ n − l + 1,   i + l − 1 ≤ j ≤ n
            j ← i + l − 1;// A_{i..j}长度为l, j − i + 1 = l
            m[i, j] ← ∞;
            for k ← i  to  j − 1  do{//分裂点k
                q ← m[i,k] + m[k+1, j] + p_{i−1} p_k p_j;
                if q < m[i, j]  then{
                    m[i, j] ← q;
                    S[i, j] ← k;
                }//endif
            }//endfor k
        }//endfor i
    }//endfor l
    return  m & S;
```

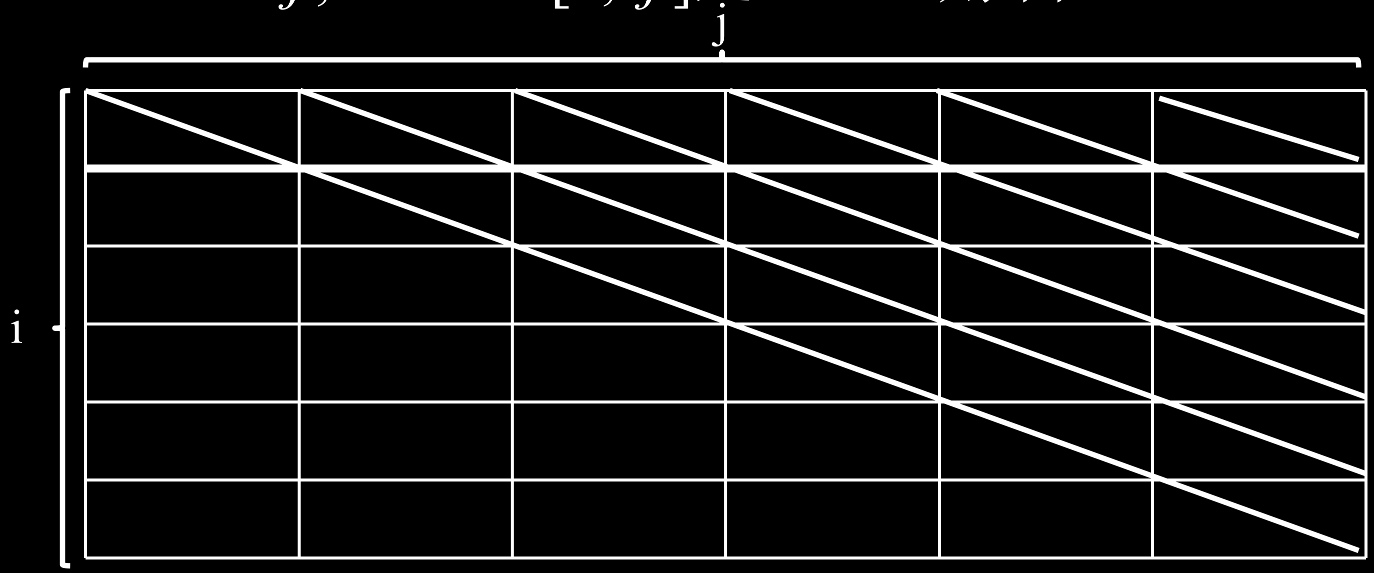
$l=2$:先计算$m[1,2],m[2,3],\ldots,m[n-1,n]$

$l=3$:先计算$m[1,3],m[2,4],\ldots,m[n-2,n]$

- **例:** $A_{30\times35}\,A_{35\times15}\,A_{15\times5}\,A_{5\times10}\,A_{10\times20}\,A_{20\times25}$

  $p=<30,35,15,5,10,20,25>$

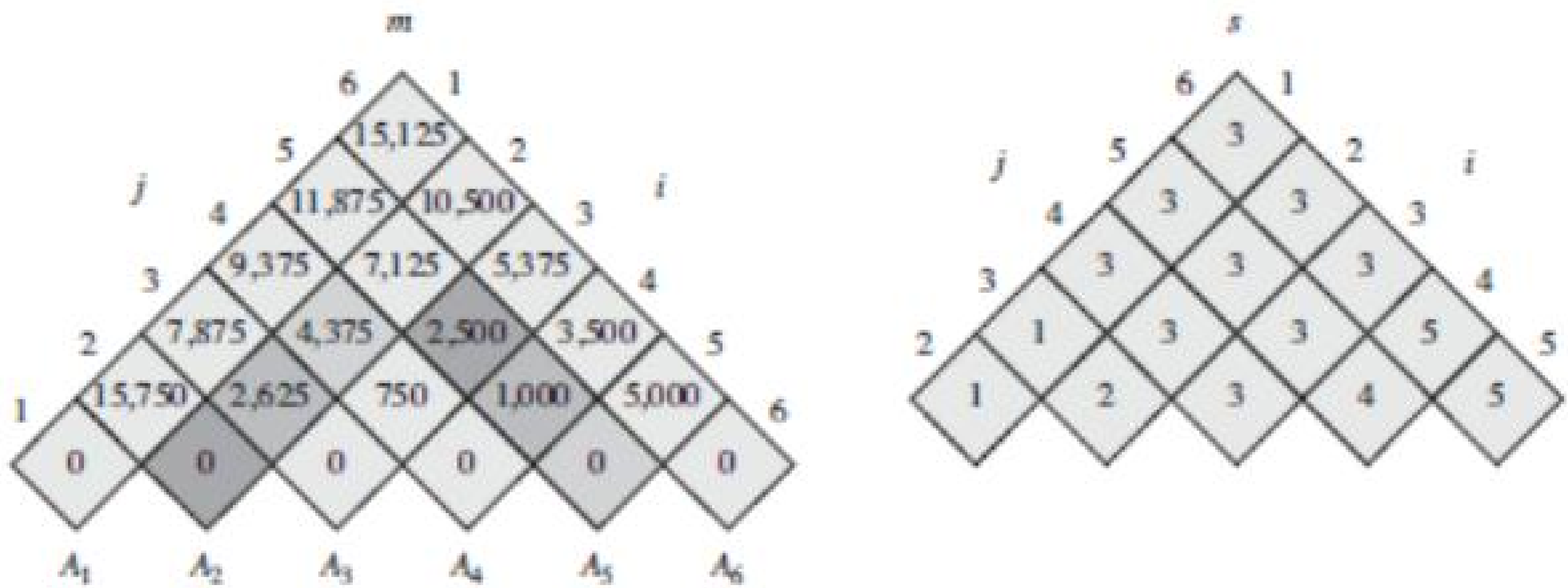
  $\because i\le j,\quad \therefore m[i,j]$是上三角阵



$T(n)=O(n^3)$, 非指数。

$S(n)=\Theta(n^2)$

■ **例：**



$$m[2,5] = \min\begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35\cdot 15\cdot 20 & = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35\cdot 5\cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35\cdot 10\cdot 20 & = 11{,}375 \end{cases}$$
$$= 7125\,.$$

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

矩阵链乘

- **Step4：构造一个最优解**

∵S[i,j]=k记录了$A_{i..j}$最优括号化分裂点为k,

∴设k_1=S[1,n]，则$A_{1..n}$的计算次序是$A_{1..k_1} \cdot A_{k_1+1..n}$

而 $A_{1..k_1}$的计算次序应为：$A_{1..k_2} \cdot A_{k_2+1..k_1}, k_2 = S[1, k_1]$

$A_{k_1+1..n}$的计算次序应为：

$$A_{k_1+1..k_3} \cdot A_{k_3+1..n}, k_3 = S[k_1+1, n]$$

一般地，$A_{i..j}$的分裂点为S[i,j]=k

$$A_{i..j} \Rightarrow A_{1..k} \cdot A_{k+1,j}$$

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i=j
   then print "A"_i
   else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j]+1, j)
        print ")"
```

动态规划要素

- 什么样的优化问题适合使用动态规划？
  - 最优子结构
  - 重叠子问题

利用重叠子问题特性可导出动态规划的变种方法：memoization方法

动态规划要素

- Optimal Substructure
- 细节
- 重叠子问题
- 重构最优解
- Memoization

Optimal Substructure

- 若一个问题的最优解，其内部包含的所有子问题解也必须最优，则该问题呈现了“最优子结构”，具有此结构特征的问题可能会使用动态规划。
  - 在动态规划中，可用子问题的最优解来构造原问题的最优解
  - 例如：矩阵$A_i A_{i+1} \cdots A_j$ 的最优括号化问题蕴含着两个子问题 $A_{i..k}$ 和 $A_{k+1..j}$ 的解也必须是最优的

Optimal Substructure

- 如何发现最优子结构呢？
  - 说明问题的解必须进行某种选择，这种选择导致一个或多个待解的子问题
  - 对一给定问题，假定导致最优解的选择已给定，即无须关心如何做出选择，只须假定它已给出. 给定选择后，决定由此产生哪些子问题，如何最好地描述子问题空间
  - 证明用在问题最优解内的子问题的解也必须是最优的。方法是“cut-and-paste”技术和反证法。假定在最优解内子问题的解非最优，删去它换上最优解，得到原问题的解非最优，矛盾！

Optimal Substructure

- 如何描述子问题空间(子问题结构,不同的子问题个数)

  尽可能使其简单，然后再考虑有没有必要扩展. 例:

  $$A_{1..n} \Rightarrow A_{1..k} \cdot A_{k+1..n} \Rightarrow (A_{1..k_1} \cdot A_{k_1+1..k})(A_{k+1..k_2} \cdot A_{k_2+1..n})$$

  由此可见，最合适的子问题空间描述为：$A_i A_{i+1} \cdots A_j$

- 最优子结构有关的两方面问题
  - 用在最优解中有多少个子问题
  - 用在最优解中的子问题有多少种选择

    例如，矩阵链乘 $A_i A_{i+1} \cdots A_j$ ——两个子问题，j-i 种选择

Optimal Substructure

- 动态规划算法的运行时间
  - 子问题总数
  - 对每个子问题涉及多少种选择

  例如:

  矩阵链乘共要解$\Theta(n^2)$个子问题: 1≤i≤j≤n

  求解每个子问题至多有n-1种选择

  最终的运行时间为 $\Theta(n^3)$
- 动态规划求解方式
  - 自底向上

细节

当心不要随便假定最优子结构的应用

例如有向无权图中，求最短/长路径的问题(指简单路径)

- 最短路径含有最优子结构

设从u到v的最短路径是P，并设中间点为w，则

$$u \xrightarrow{P} v \quad \Rightarrow \quad u \xrightarrow{P_1} w \xrightarrow{P_2} v$$

显然P1和P2也必须是最优（短）的。

细节

- 最长路径不具有最优子结构

设P 是从u到v的最长路径，w是中间某点，则

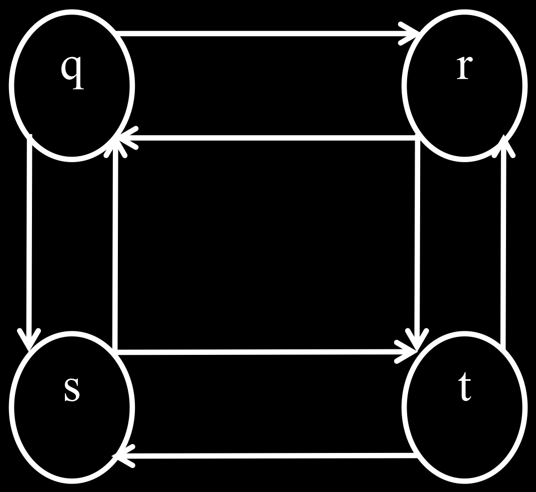
$$u \xrightarrow{P} v \Rightarrow u \xrightarrow{P_1} w \xrightarrow{P_2} v$$

但P1不是从u到w的最长路径，P2也不是从w到v的最长路径。 例：

考虑一最长路径 $q \to r \to t$

但q到r的最长路径是：$q \to s \to t \to r$

r到t的最长路径是：$r \to q \to s \to t$



细节

- 为什么两问题有差别？
  - 最长路径的子问题不是独立的。所谓独立指一个子问题的解不能影响另一个子问题的解。但第一个子问题中使用了s和t，第二个子问题又使用了，使得产生的路径不再是简单路径。

    从另一个角度看一个子问题求解时使用的资源（顶点）不能在另一个子问题中再使用。
  - 最短路径问题中，两子问题没有共享资源，可用反证法证明之。

    例：矩阵链乘 $A_{i..j} \Rightarrow A_{i..k} \cdot A_{k+1..j}$

显然两子链不相交，没有资源共享，是相互独立的两子问题

重叠子问题

当用递归算法解某问题时，重复访问（计算）同一子问题

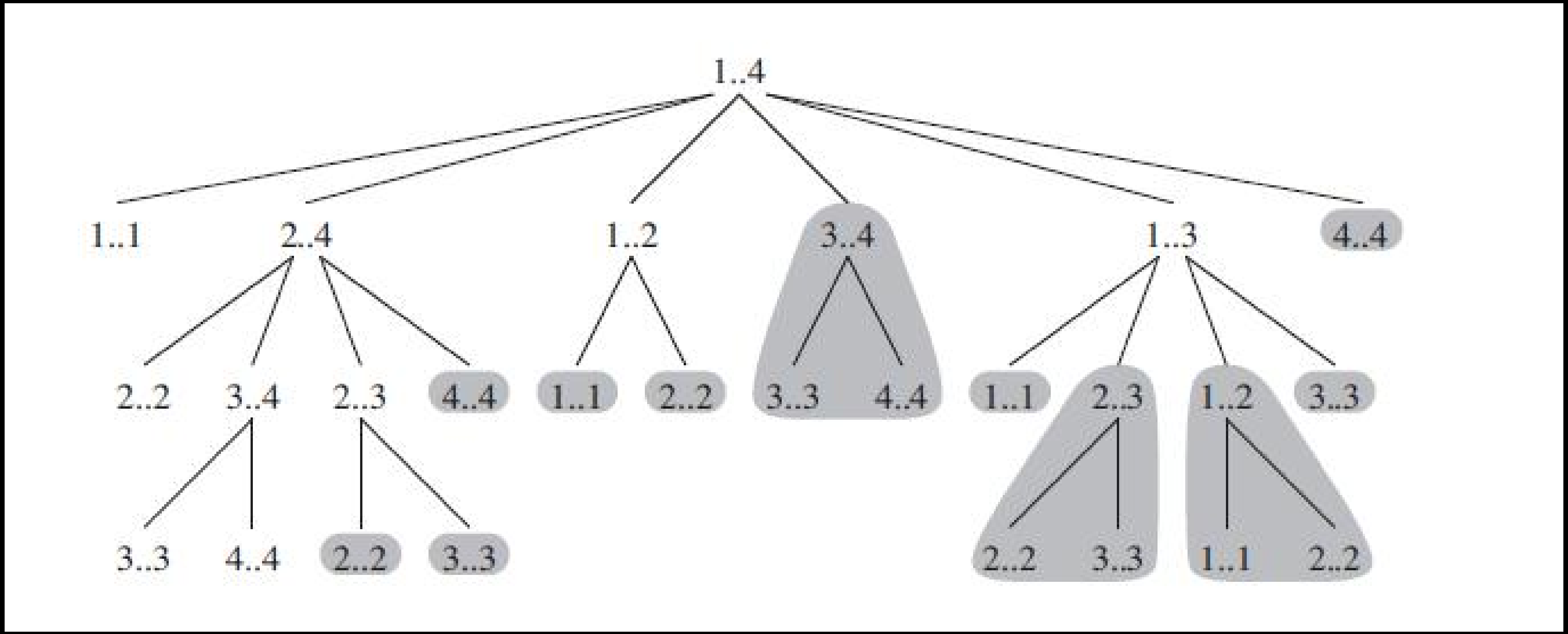
- 分治法与动态规划的比较
  - 当递归每一步产生一个新的子问题时，适合使用分治法
  - 当递归中较多出现重叠子问题时，适合使用动态规划，即对重叠子问题只求解一次，然后存储在表中，当需要使用时常数时间内查表。若子问题规模是多项式阶的，动态规划特别有效。

重叠子问题

- 例：用自然递归算法求解

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

时 $A_{1\dots 4}$ 的递归树为：



阴影部分是重叠子问题，递归算法须重复计算。

重叠子问题

时间：

$$\begin{cases} T(1) \geq 1 \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \end{cases} \Rightarrow T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n$$

用代入法可证为 $\Omega(2^n)$

- **结论：当自然递归算法是指数阶，但实际不同的子问题数目是多项式阶时，可用动态规划来获得高效算法**

重构最优解

- 用附加表保存中间选择结果能节省重构最优解的时间

Memoization

- 动态规划：分析是自顶向下，实现是自底向上
- 可采用备忘（记忆）型版本。它是动态规划的变种，效率和动态规划相似，但采用自顶向下实现，故是一个记忆型递归算法。
- 和动态规划类似，将子问题的解记录在一个表中，但填表的控制结构更像递归算法，其特点是：
  - 每个子问题的解对应一表项
  - 每个表目初值唯一，特殊值表示尚未填入
  - 在递归算法执行过程中第一次遇某子问题时，计算其解并填入表中，以后再遇此子问题时，将表中值简单地返回（不重复计算），截断递归。
- 该方法的前提
  - 原有可能的子问题参数集合是已知的
  - 可在表位置和子问题间建立某种关系

$$
\begin{aligned}
&MemoizedMatrixChain(p)\{\\
&\quad n \leftarrow length[p]-1;\\
&\quad for\ \ i=1 \quad to \quad n \quad do\\
&\qquad for\ j=i \quad to \quad n \quad do\\
&\qquad\quad m[i,j] \leftarrow \infty;\\
&\qquad\quad //\text{表目初值，上三角}\\
&\quad return \qquad LookupChain(p,1,n);\\
&\}
\end{aligned}
$$

```
LookupChain(p, i, j){
    if m[i, j] < ∞   then //已计算过
        return   m[i, j]; //截断递归
    //第一次遇到子问题A_{i..j}， 计算之
    if i = j   then
        m[i, j] = 0;
    else
        for k ← i   to   j − 1   then{
            q ← LookupChain(p, i, k) +
                LookupChain(p, k + 1, j) + p_{i−1} p_k p_j;
            if q < m[i, j]   then   m[i, j] ← q;
        }
    return   m[i, j];
}
```

初始化$\Theta(n^2)$, $\Theta(n^2)$个表目每个仅计算一次，

但计算一个表目时需要$O(n)$时间，故总共$\Theta(n^3)$

小结

- 若所有子问题须至少解一次，自底向上的动态规划时间常数因子较优（不需要递归开销，维护表的开销较小）
- 若子问题空间有些不需要计算，则备忘型递归具有只需计算需要的子问题的优点。

思考题

例8 最大子段和

问题：给定n个整数（可以为负数）的序列

$(a_1, a_2, \ldots, a_n)$

求

$$\max\{\ 0,\ \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k\}$$

实例： (-2, 11, -4, 13, -5, -2)

解：最大子段和 $a_2+a_3+a_4= 20$

算法1---顺序求和+比较

算法2---分治策略

算法3---动态规划

算法2　分治策略

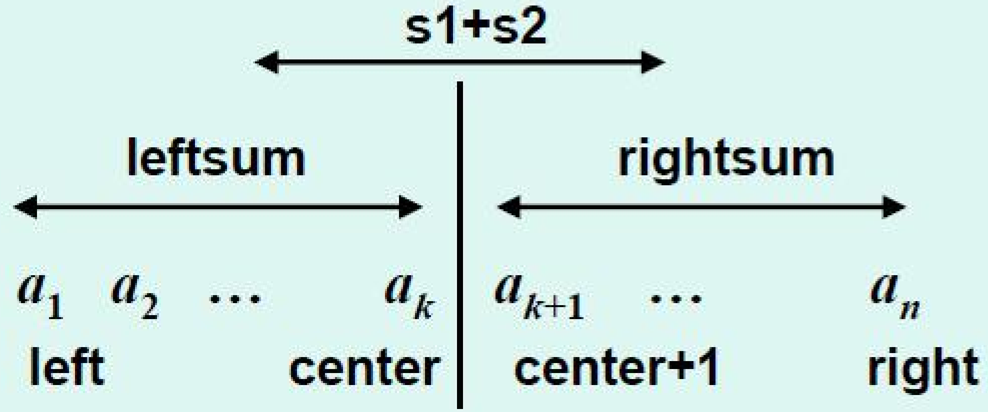
将序列分成左右两半，中间分点*center*

递归计算左段最大子段和 *leftsum*

递归计算右段最大子段和 *rightsum*

$a_{center} \to a_1$的最大和$s1$，$a_{center+1} \to a_n$的最大和$s2$

max { *leftsum*, *rightsum*, $s1+s2$}



动态规划算法 MaxSum

令 $b[j]=\max\limits_{1\le i\le j}\{\sum\limits_{k=i}^{j}a[k]\}$

多步判断：

$b[j]$表示最后一项为$a[j]$的序列构成的最大的子段和

最优解为$b[1], b[2], ..., b[n]$中的最大值

递推方程为

$$b[j]=\max\{b[j-1]+a[j],a[j]\} \quad j=1,2，\ldots,n$$