第3章　动态规划

学习要点:

- 理解动态规划算法的概念。
- 掌握动态规划算法的基本要素。
- 掌握设计动态规划算法的步骤
- 通过范例学习动态规划算法设计策略

分治法 VS 动态规划法 Divide and conquer VS Dynamic Programming

分治法：分而治之，治而合之
动规法：重叠子问题，最优子结构

动态规划（Dynamic Programming）

- **动态规划算法的概念**
- 动态规划算法的基本要素
  - 最优子结构性质
  - 重叠子问题性质
- 设计动态规划算法的步骤
  - 找出最优解的性质，并刻划其结构特征
  - 递归地定义最优值。
  - 以自底向上的方式计算出最优值。
  - 根据计算最优值时得到的信息，构造最优解

动态规划算法的基本思想

- 动态规划方法：处理分段过程最优化问题的一类极其有效的方法。
- 多阶段的决策过程
  - 问题的活动过程可以分成若干个阶段
  - 在任一阶段后的行为依赖于该阶段的状态
  - 与该阶段之前的过程是如何达到这种状态的方式无关（无后效性、马尔可夫性质）。

未来与过去无关！

往者不可谏，来者犹可追

动态规划的历史（1/5）

有许多问题，用穷举法才能得到最佳解。若输入量n稍大一些，计算量太大，特别对渐近时间复杂性为输入量的指数函数的问题，计算机无法完成。采用动态规划（Dynamic programming）能得到比穷举法更有效的算法。动态规划的指导思想是，在每种情况下，列出各种可能的局部解，从局部解中挑出那些有可能产生最佳的结果而扬弃其余，从而大大缩减了计算量。

动态规划的历史（2/5）

动态规划遵循的“最佳原理”简而言之，“一个最优策略的子策略总是最优的”。动态规划是运筹学的一个分支，它是解决多阶段决策过程最优化的一种方法，大约产生于50年代，由美国数学家贝尔曼（R·Bellman）等人，根据一类多阶段决策问题的特点，把多阶段决策问题变换为一系列互相联系单阶段问题，然后逐个加以解决。

动态规划的历史（3/5）

动态规划开始只是应用于多阶段决策性问题，后来渐渐被发展为解决离散最优化问题的有效手段，进一步应用于一些连续性问题上。然而，动态规划更像是一种思想而非算法，它没有固定的数学模型，没有固定的实现方法，其正确性也缺乏严格的理论证明。因此，一直以来动态规划的数学理论模型是一个研究的热点。

动态规划的历史（4/5）


贝尔曼，R.

贝尔曼，R Richard Bellman (1920～1984)

美国数学家，美国科学院院士，动态规划的创始人。1920年8月26日生于美国纽约。1984年3月19日逝世。1941年在布鲁克林学院毕业，获理学士学位，1943年在威斯康星大学获理学硕士学位，1946年在普林斯顿大学获博士学位。1946～1948年在普林斯顿大学任助理教授，1948～1952年在斯坦福大学任副教授，1953～1956年在美国兰德公司任研究员，1956年后在南加利福尼亚大学任数学教授、电气工程教授和医学教授。

贝尔曼因提出动态规划而获美国数学会和美国工程数学与应用数学会联合颁发的第一届维纳奖金（1970），卡内基－梅隆大学颁发的第一届迪克森奖金(1970)，美国管理科学研究会和美国运筹学会联合颁发的诺伊曼理论奖金(1976)。1977年贝尔曼当选为美国艺术与科学研究院院士和美国工程科学院院士。

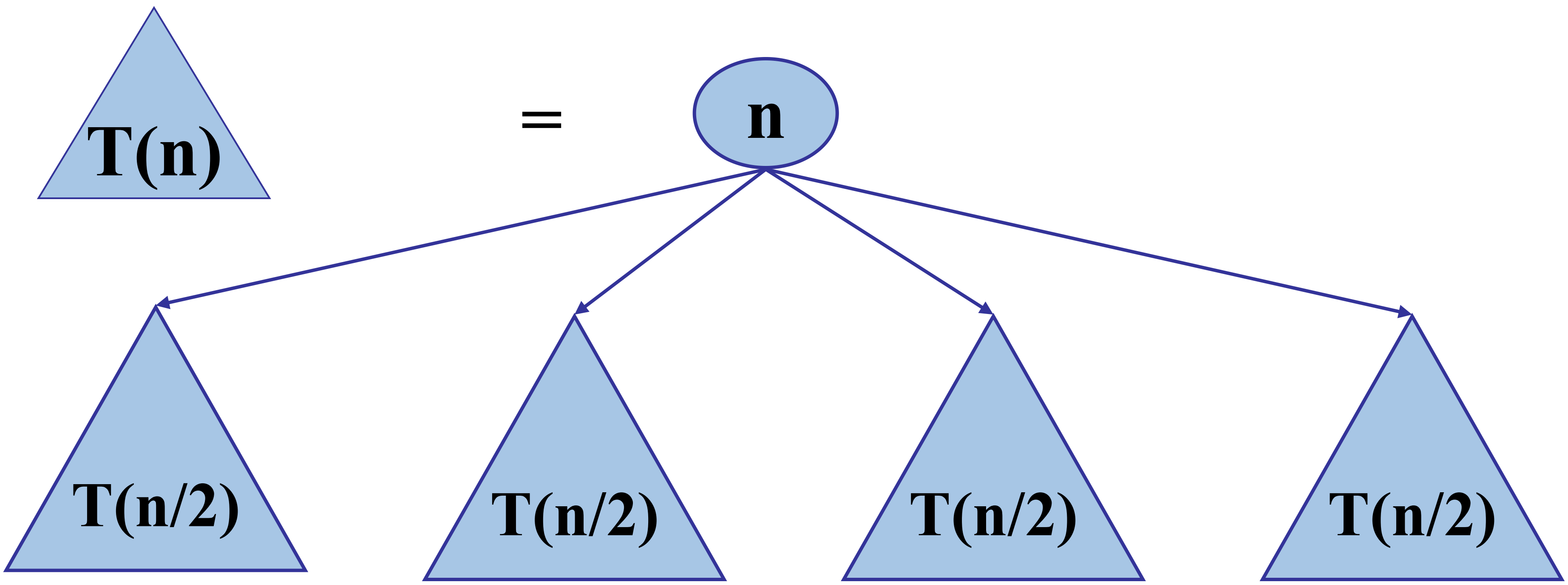
动态规划的历史（5/5）

贝尔曼因在研究多段决策过程中提出动态规划而闻名于世。1957年他的专著《动态规划》出版后，被迅速译成俄文、日文、德文和法文，对控制理论界和数学界有深远影响。贝尔曼还把不变嵌入原理应用于理论物理和数学分析方面，把两点边值问题化为初值问题，简化了问题的分析和求解过程。1955年后贝尔曼开始研究算法、计算机仿真和人工智能，把建模与仿真等数学方法应用到工程、经济、社会和医学等方面，取得许多成就。贝尔曼对稳定性的矩阵理论、时滞系统、自适应控制过程、分岔理论、微分和积分不等式等方面都有过贡献。

贝尔曼曾是《数学分析与应用杂志》及《数学生物科学杂志》的主编，《科学与工程中的数学》丛书的主编。已出版30本著作和7本专著，发表了600多篇研究论文。

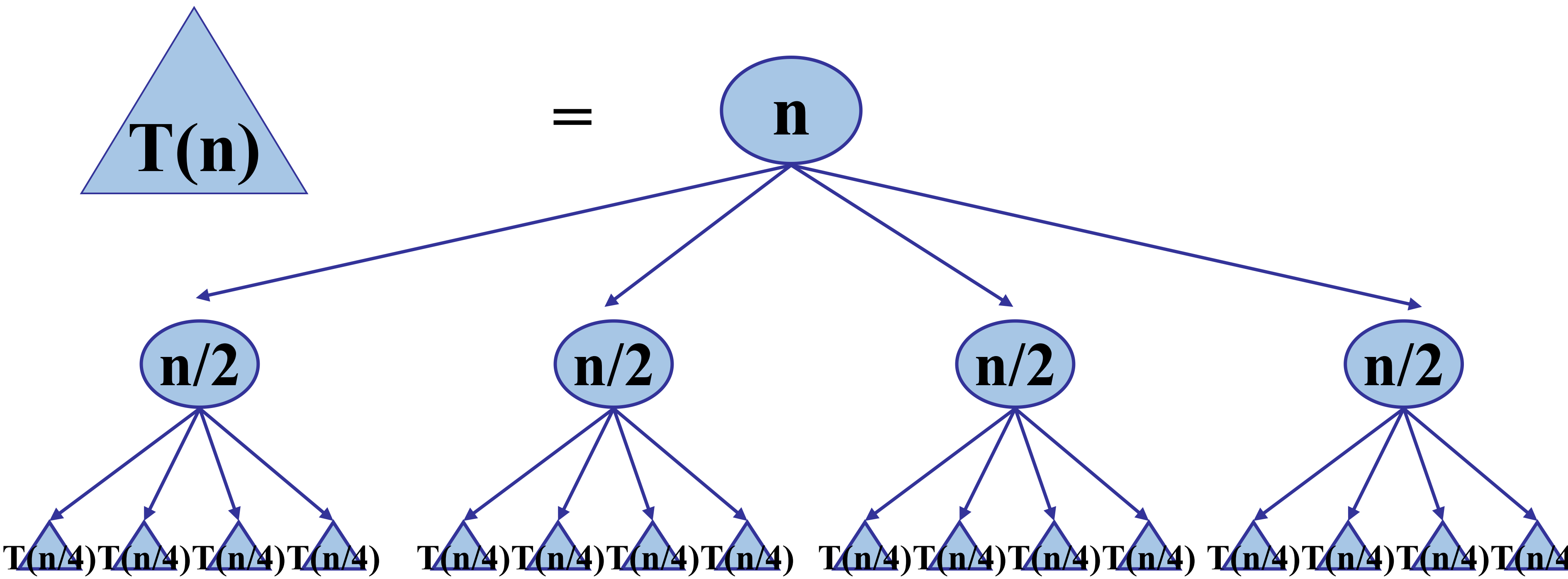
算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



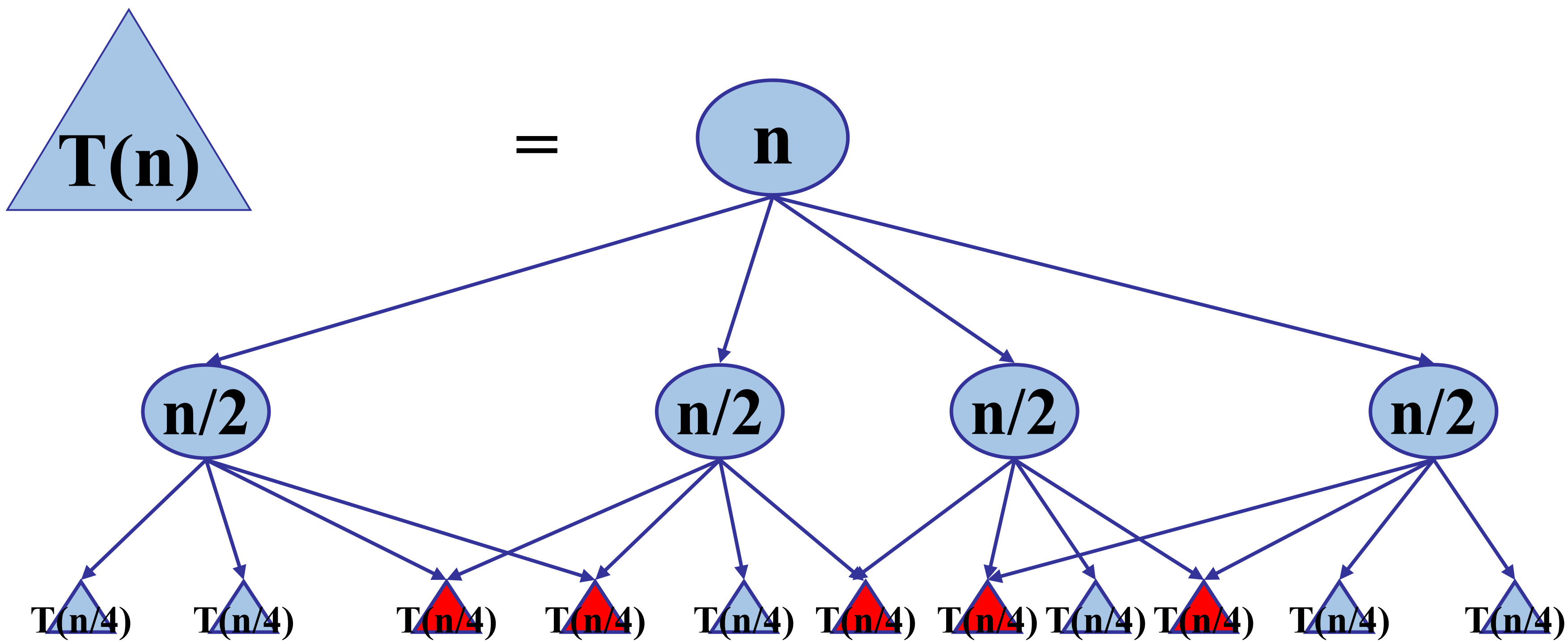
算法总体思想

- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。



算法总体思想

- 解决方法：保存已解决的子问题的答案，在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。



动态规划基本步骤

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

动态规划算法的基本要素(1)
- 最优子结构

- 最优子结构性质：问题的最优解包含着其子问题的最优解。
- 证明-反证法：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 运算方向：利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。
- 最优子结构是问题能用动态规划算法求解的前提。

同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快（空间占用小，问题的维度低）

动态规划算法的基本要素(2)
-重叠子问题

- 子问题的重叠性质:递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。
- 动态规划算法，对每一个子问题只解一次，将其解保存在一个表格中，当需要再次解此子问题时，用常数时间查看结果。
- 不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。

引导例子（1/10）

最短路径问题描述：

输入：起点集合｛ S_1， S_2， . . . ， S_n ｝ ，

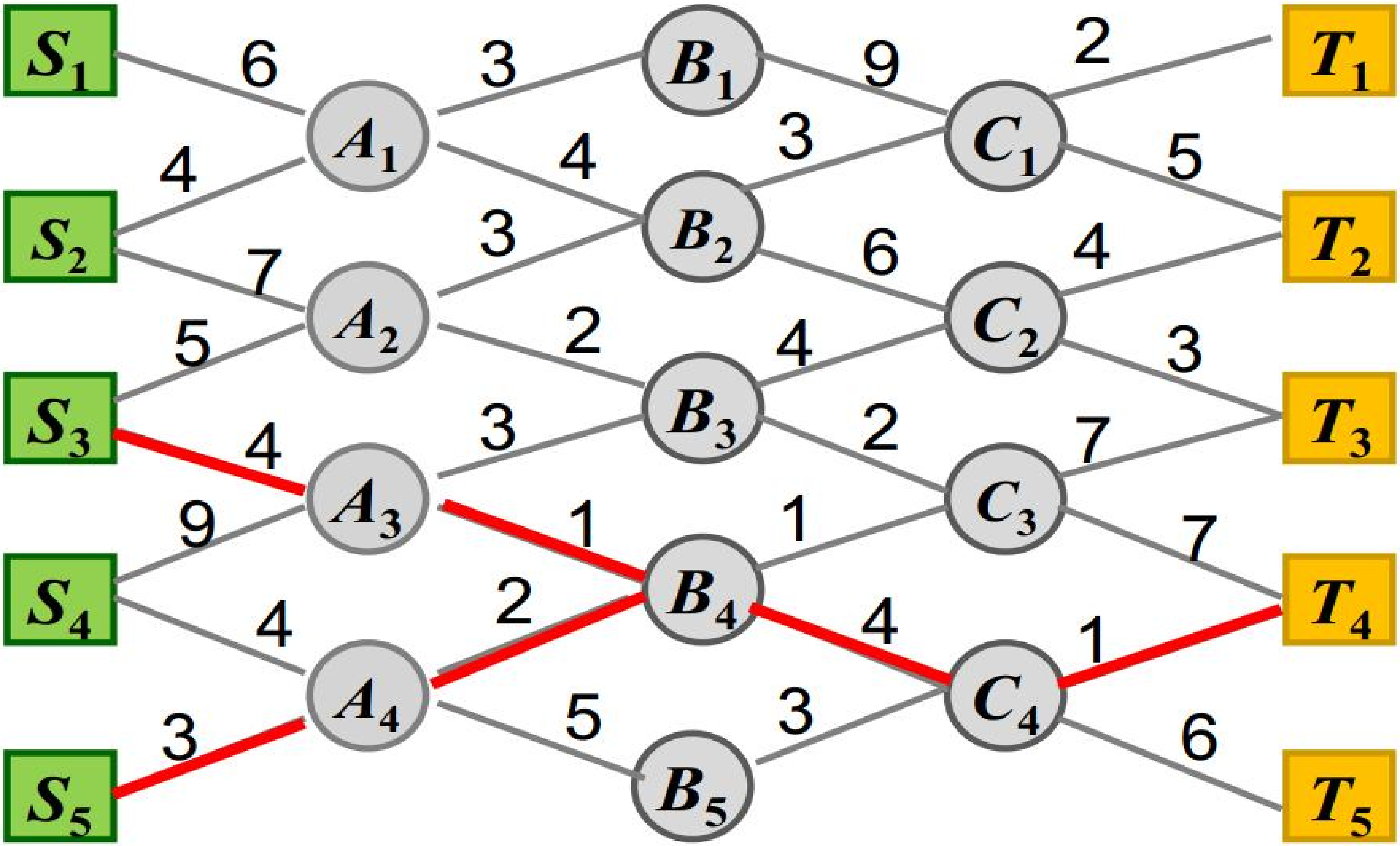
终点集合｛T_1， T_2， . . . ， T_m｝，

中间结点集，

边集E，对于任意边e有长度

输出：一条从起点到终点的最短路

引导例子(2/10)



S_1
S_2
S_3
S_4
S_5
A_1
A_2
A_3
A_4
B_1
B_2
B_3
B_4
B_5
C_1
C_2
C_3
C_4
T_1
T_2
T_3
T_4
T_5
6
4
7
5
4
9
4
3
3
4
3
2
3
1
2
5
9
3
6
4
2
1
4
3
2
5
4
3
7
7
1
6

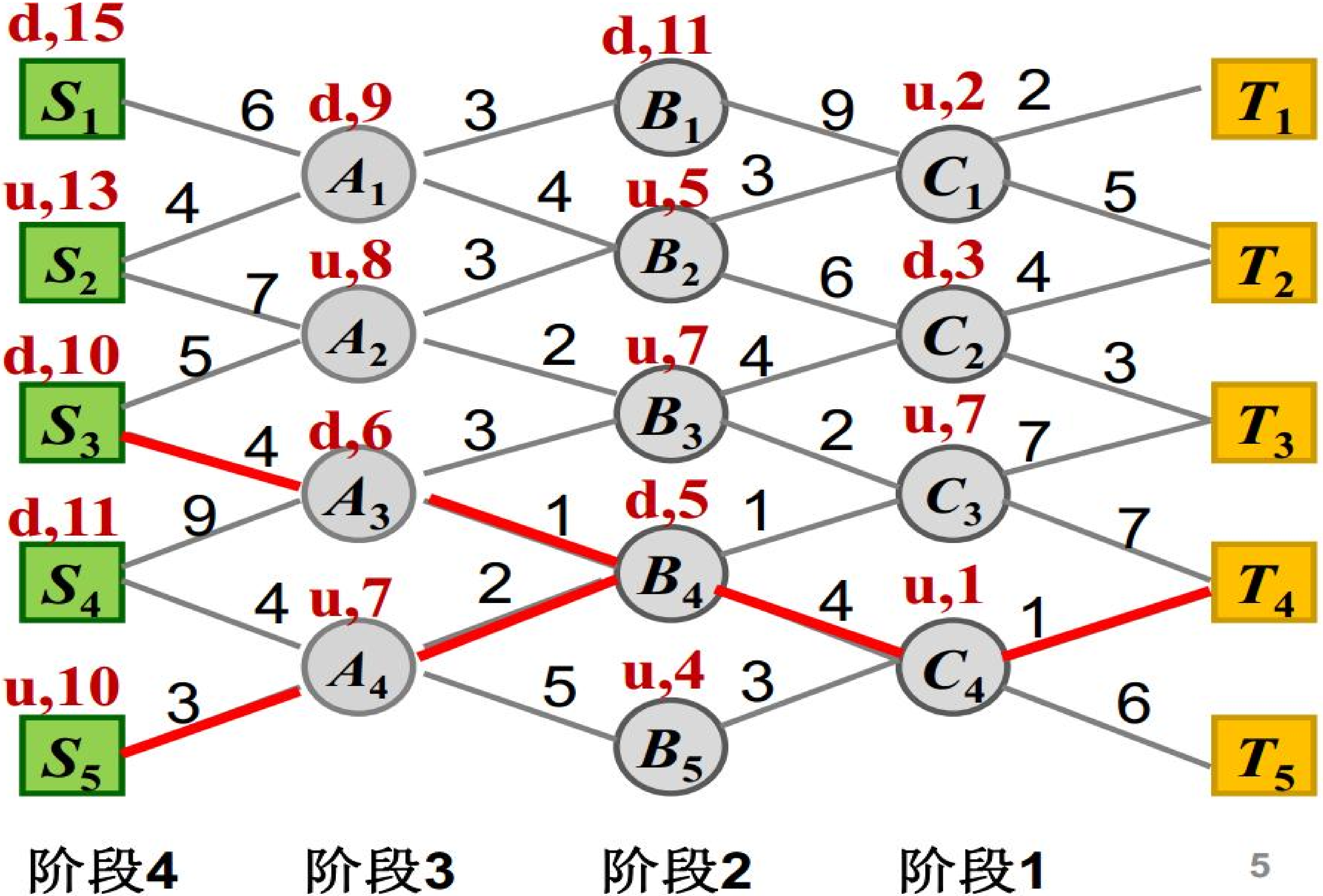
引导例子(3/10)

蛮力算法：考察每一条从某个起点到某个终点的路径，计算长度，从其中找出最短路径。

在上述实例中，如果网络的层数为k，那么路径条数将接近于2^K。

动态规划算法：多阶段决策过程.每步求解的问题是后面阶段求解问题的子问题. 每步决策将依赖于以前步骤的决策结果.

引导例子(4/10)



d,15
S_1
u,13
S_2
d,10
S_3
d,11
S_4
u,10
S_5
d,9
A_1
u,8
A_2
d,6
A_3
u,7
A_4
d,11
B_1
u,5
B_2
u,7
B_3
d,5
B_4
u,4
B_5
u,2
C_1
d,3
C_2
u,7
C_3
u,1
C_4
T_1
T_2
T_3
T_4
T_5
阶段4
阶段3
阶段2
阶段1

引导例子(5/10)

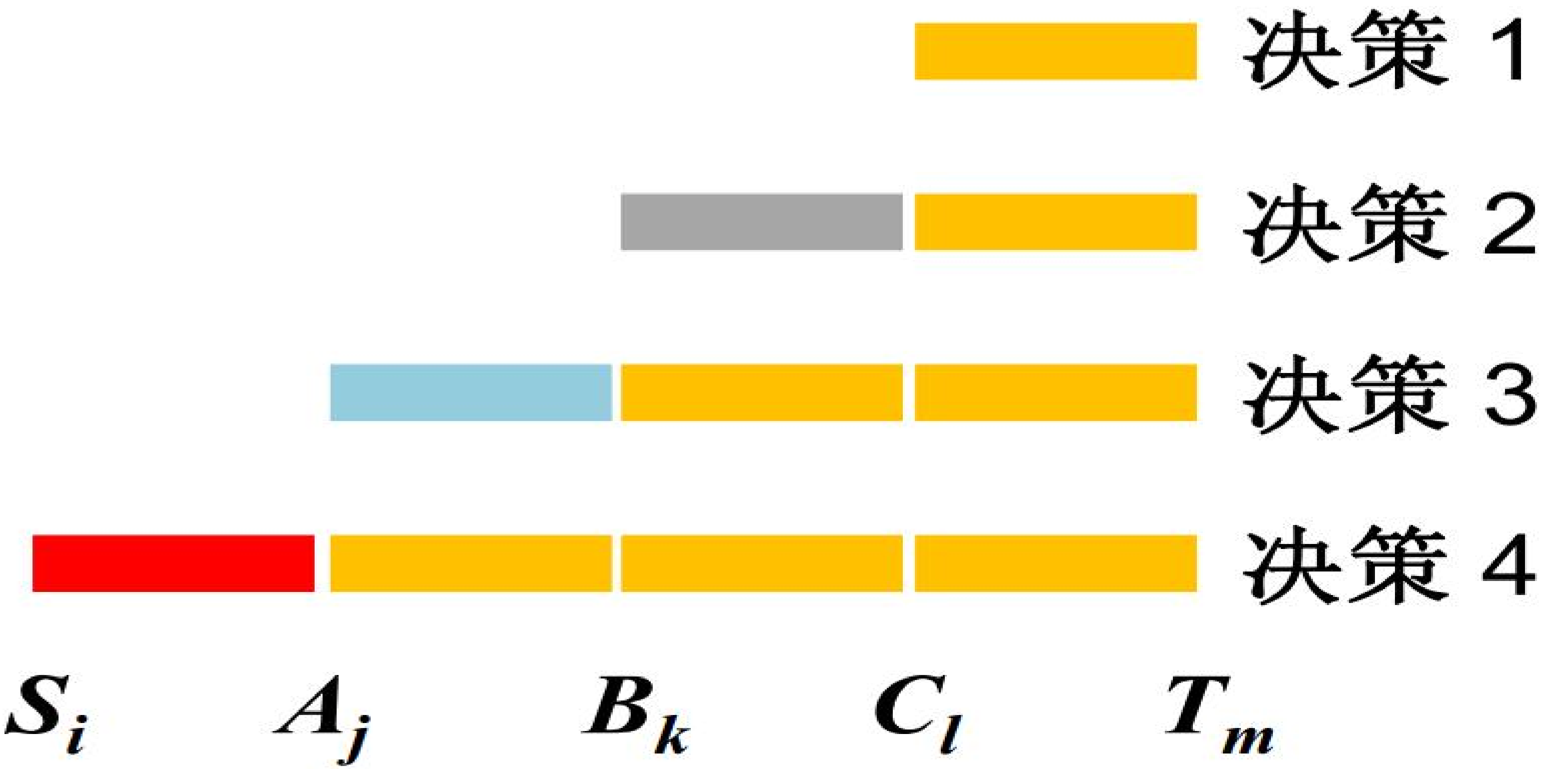
蛮力算法：考察每一条从某个起点到某个终点的路径，计算长度，从其中找出最短路径。

在上述实例中，如果网络的层数为k，那么路径条数将接近于2^K。

动态规划算法：多阶段决策过程. 每步求解的问题是后面阶段求解问题的子问题. 每步决策将依赖于以前步骤的决策结果.

引导例子(6/10)

前边界不变，后边界前移



引导例子(7/10)

$$F(C_l)=\min_m\{C_lT_m\}$$ 决策 1

$$F(B_k)=\min_l\{B_kC_l+F(C_l)\}$$ 决策 2

$$F(A_j)=\min_k\{A_jB_k+F(B_k)\}$$ 决策 3

$$F(S_i)=\min_j\{S_iA_j+F(A_j)\}$$ 决策 4

优化函数值之间存在依赖关系

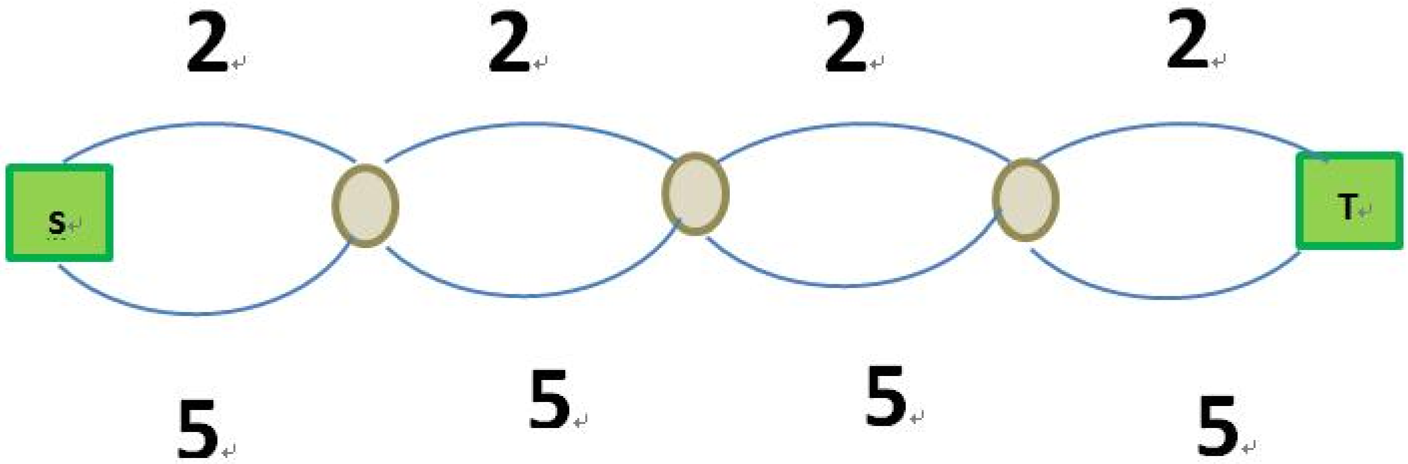
引导例子(8/10)

优化函数的特点：任何最短路的子路径相对于子问题始、终点最短

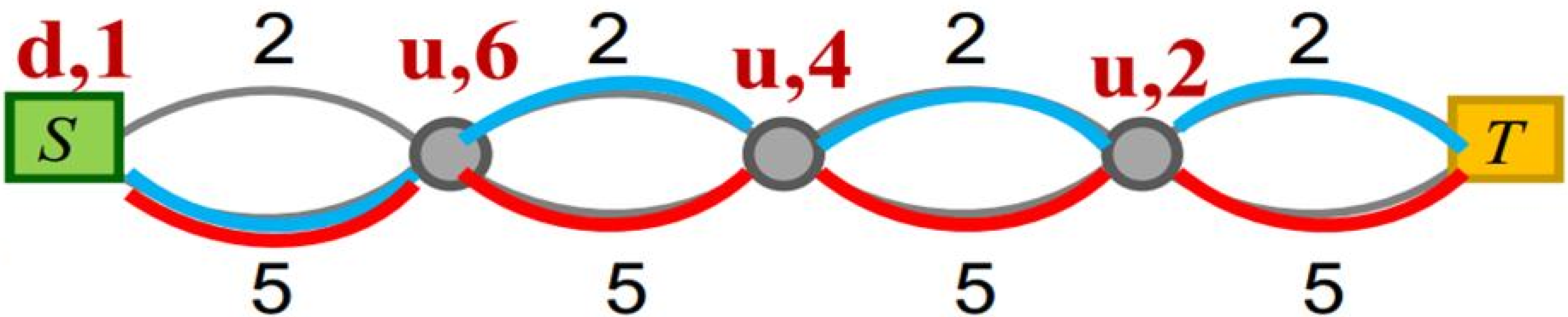
优化原则：一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列

引导例子(9/10)

求总长模10的最小路径



引导例子(10/10)



动态规划算法的解：下, 上, 上, 上

最优解：下, 下, 下, 下

不满足优化原则，不能用动态规划

§3.1 矩阵连乘问题

给定n个矩阵{A_1， A_2， …， A_n}，其中A_i与A_{i+1} 是可乘的。考察这N个矩阵的连乘积 $A_1A_2\ldots A_n$

分析：

由于矩阵乘法满足结合律，故计算矩阵连乘积$A_1A_2\ldots A_n$可以有多个计算次序。

例如：

$$\left\{\begin{array}{l}(A_1(A_2(A_3A_4))) \\ (A_1((A_2A_3)A_4)) \\ ((A_1A_2)(A_3A_4))\end{array}\right.$$

等等

先考虑两个矩阵乘积的计算量

设A为ra×ca矩阵 B为rb×cb矩阵，

```
void  matrixMultiply(int **a, int **b,  int **c,
                          int ra,  int ca,  int rb, int cb )
{   if(ca!=rb) error( "Can't multiply" );
    for(i=0; i<ra; ++i)
      for(j=0; j<cb; ++j)
        {    c[i][j] =0;
             for(k=0;k<ca; ++k)  c[i][j]+=a[i][k]*b[k][j];
}        }
```

矩连连乘的不同计算次序会导致不同的计算量

例如 $\{A_1, A_2, A_3\}$　　A_1　10*100

A_2　100*5

A_3　5*50

第1种加括号

$((A_1A_2)A_3)$　　10*100*5

+10*5*50　　=7500

第2种加括号

$(A_1(A_2A_3))$　100*5*50

+10*100*50　=75000

所谓矩阵连乘问题：

对于给定n个矩阵{A_1， A_2， …， A_n}，

其中A_i的维数是$p_{i-1} \times p_i$

如何确定计算矩阵连乘积$A_1A_2 \ldots A_n$的计算次序，使得计算矩阵连乘积的数乘次数最少。

方法一：

枚举所有可能的计算次序。 $\Omega(2^n / n^{3/2})$

方法二：

动态规划。 $O(n^3)$

第一种：矩阵连乘的递归求解方法

记 $A[i:j]$为 $A_iA_{i+1}\ldots A_j$　考察计算$A[1:n]$的最优计算次序

不妨设

计算$A[1:n]$最优次序的最后一次乘积位置在 K 处

$$(\underline{(A_1A_2\ldots A_k)}\ \underline{(A_{k+1}\ldots A_n)})\qquad k=1, 2, \ldots n-1$$

其计算量为

(1)计算 $A[1:k]$

(2)计算 $A[k+1:n]$

(3)最后一次矩阵乘积$p_0\times p_k\times p_n$

1.分析最优解的结构

计算A[1:n]的最优次序所包含的子链计算

A[1:k] 和 A[k+1:n]也一定是最优次序的。

事实上

若有一个计算A[1:k]的次序所需的计算量更少，则取代之。类似，计算A[k+1:n]的次序也一定是最优的。

因此，矩阵连乘计算次序问题的最优解，包含了其子问题的最优解。

2. 建立递归关系，定义最优值

设计算A[i:j]所需的最少数乘次数为m[i][j]

则原问题的最优值为m[1][n]

不妨设

计算A[i:j]最优次序的最后一次乘积位置在 K 处

$((A_iA_2\ldots A_k)(A_{k+1}\ldots A_j))$ $k=i,2,\ldots j-1$

则

$$m[i][j]=\begin{cases} 0 & i=j \\ \underset{i\le k<j}{\mathrm{MIN}}\{m[i][k]+m[k+1][j]+p_{i-1}p_kp_j\} & i<j \end{cases}$$

i<=k<j

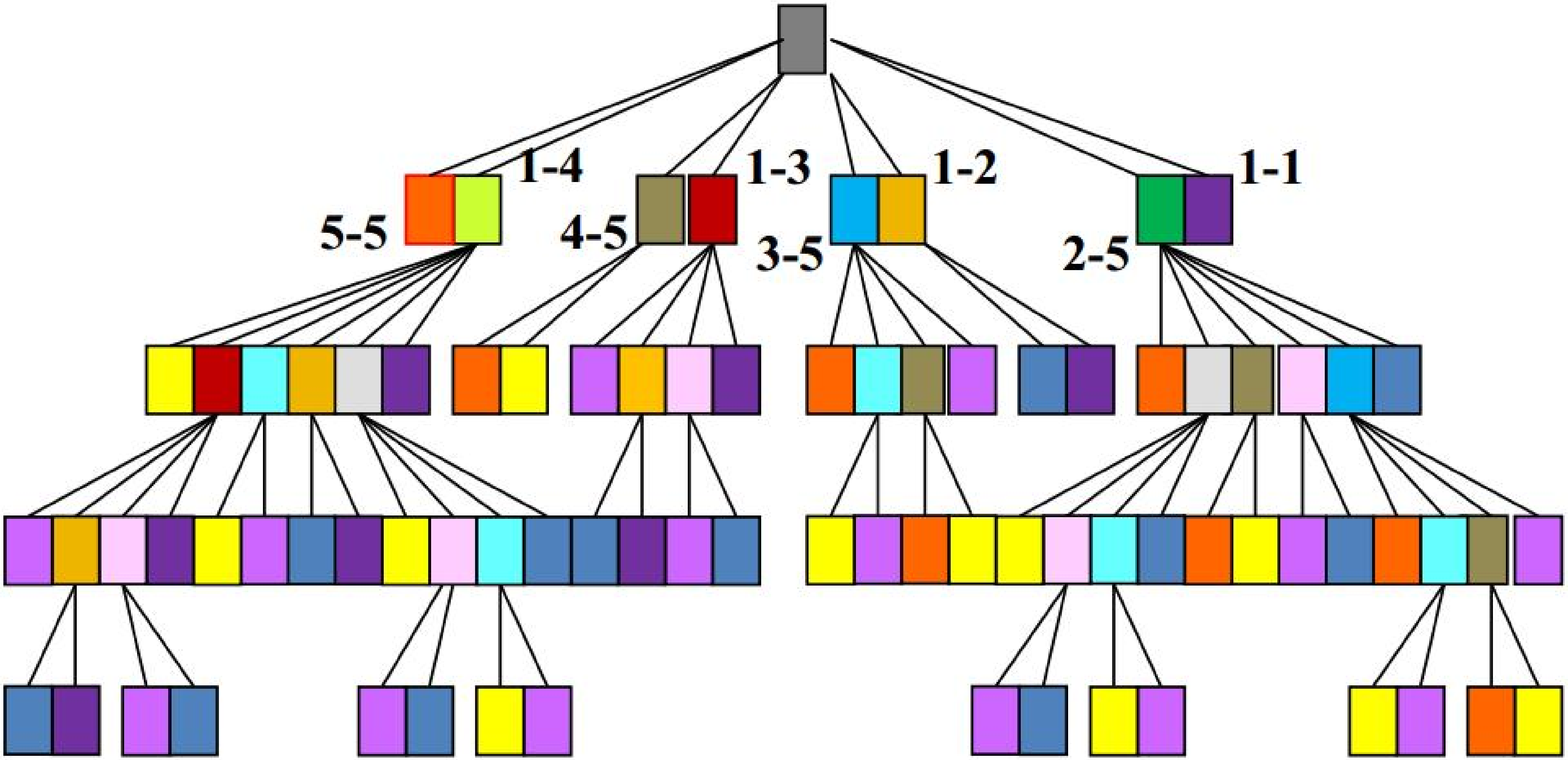
3. 伪码分析

算法1 RecurMatrixChain(P, i, j)

```
m[i, j]←∞
s[i, j]←i
for k←i to j−1 do
q←RecurMatrixChain(P, i, k)
   +RecurMatrixChain(P, k+1, j)+p_{i-1}p_k p_j
   if q< m[i, j]
   then m[i, j]←q
        s[i, j]←k
return m[i, j]
```

这里没有写出算法的全部描述（递归边界）

4. 子问题的产生 n=5



5. 子问题的计数

边界	次数	边界	次数	边界	次数
1-1	8	1-2	4	2-4	2
2-2	12	2-3	5	3-5	2
3-3	14	3-4	5	1-4	1
4-4	12	4-5	4	2-5	1
5-5	8	1-3	2	1-5	1

边界不同的子问题：15个

递归计算的子问题：81个

6. 结论

- 与蛮力算法相比较，动态规划算法利用了子问题优化函数间的依赖关系，时间复杂度有所降低。
- 动态规划算法的递归实现效率不高，原因在于同一子问题多次重复出现，每次出现都需要重新计算一遍。
- 采用空间换时间策略，记录每个子问题首次计算结果，后面再用时就直接取值，每个子问题只算一次。

动态规划求解分析

问题的表示：

- 将矩阵连乘积 $A_iA_{i+1}\ldots A_j$ 简记为**A[i:j]**，这里**i≤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**Ak**和**Ak+1**之间将矩阵链断开，**i≤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

动态规划求解分析

分析最优解的结构：

特征：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。这种性质称为最优子结构性质。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。

动态规划求解分析

建立递归关系：

设计算A[i:j]，1≤i≤j≤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]

当i=j时，A[i:j]=Ai，因此，m[i,i]=0，i=1,2,…,n

当i<j时， $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$

这里 A_i 的维数为 $p_{i-1}\times p_i$

可以递归地定义m[i,j]为：

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

k 的位置只有 $j-i$ 种可能

动态规划求解分析

计算最优值：

对于**1≤i≤j≤n**不同的有序对**(i,j)**对应于不同的子问题。因此，不同子问题的个数最多只有

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

在递归计算时，**许多子问题被重复计算多次**。这也是该问题可用动态规划算法求解的又一显著特征。

用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。在计算过程中，保存已解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法。

动态规划求解方法(1/10)

动态规划实现的关键：

- 每个子问题只计算一次
- 迭代过程
  - 从最小的子问题算起
  - 考虑计算顺序，以保证后面用到的值前面已经计算好
  - 存储结构保存计算结果——备忘录
- 解的追踪
  - 设计标记函数标记每步的决策
  - 考虑根据标记函数追踪解的算法

动态规划求解方法(2/10)

矩阵连乘的不同子问题：

长度1：只含1个矩阵，有n个子问题(不需要计算)

长度2：含2个矩阵， $n-1$个子问题

长度3：含3个矩阵， $n-2$个子问题

…

长度$n-1$：含$n-1$个矩阵， 2个子问题

长度n：原始问题，只有1个

动态规划求解方法(3/10)

矩阵链乘法迭代顺序：

长度为1： 初值， $m[i, i] = 0$

长度为2： 1..2, 2..3, 3..4, ..., n–1..n

长度为3： 1..3, 2..4, 3..5, ..., n–2..n

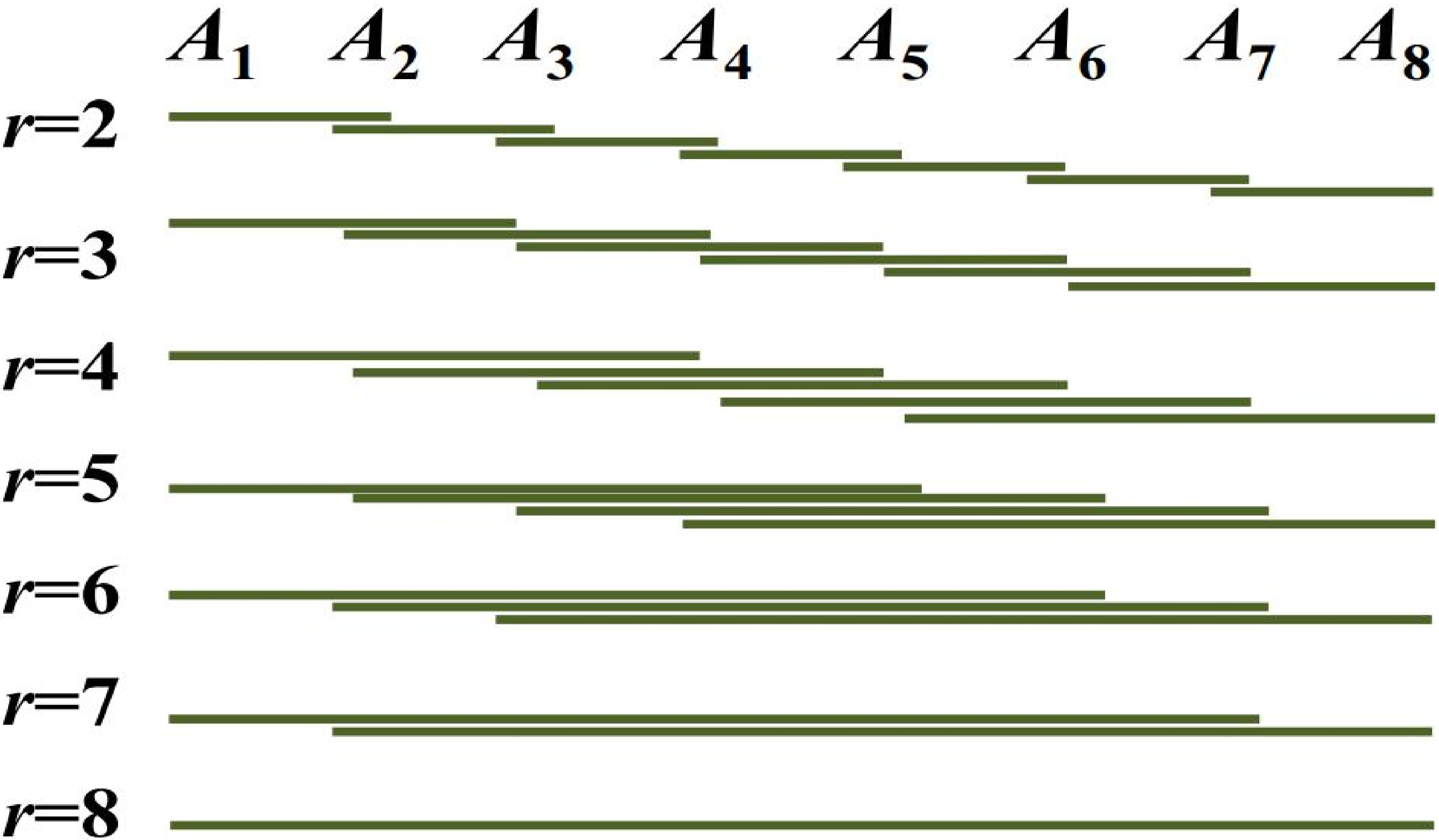
...

长度为n–1： 1..n–1, 2..n

长度为n： 1..n

动态规划求解方法(4/10)

n=8的子问题计算顺序：



动态规划求解方法(5/10)

```
void  MatricChain(int *p, int n, int **m, int **s)
{   for(i=1;i<=n; ++i)  m[i][i]=0; //单个矩阵无计算
    for(r=2; r<=n; ++r)   //连乘矩阵的个数
      for(i=1; i<n-r; ++i)
      {   j=i+r-1;
          m[i][j]=m
          s[i][j]=i;
          for(k=i+1; k<j; ++k)
         {   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;
}     }  }
```

算法复杂度分析：
算法matrixChain的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。因此算法的计算时间上界为$O(n^3)$。算法所占用的空间显然为$O(n^2)$。

动态规划求解方法(6/10)

```
void  Traceback( int i,  int j,  int **s)
{
      if (i==j)  return;
      k=s[i][j];
      Traceback(i, k, s);
      Traceback(k+1, j, s);
      printf("A[%d:%d] *A[%d:%d ] \n",  i, k, k+1, j);
}
```

动态规划求解方法(7/10)

实例分析：

输入： $P = \langle 30, 35, 15, 5, 10, 20 \rangle, n = 5$

矩阵链： $A1\ A2\ A3\ A4\ A5$，其中

$A1$：30×35，　$A2$：35×15，　$A3$：15×5，

$A4$：5×10，　$A5$：10×20

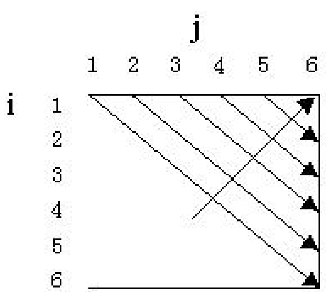
备忘录： 存储所有子问题的最小乘法次数及得到这个值的划分位置。

动态规划求解方法(8/10)

备忘录m[i,j], P=<30, 35, 15, 5, 10, 20>

r=1	*m*[1,1]=0	*m*[2,2]=0	*m*[3,3]=0	*m*[4,4]=0	*m*[5,5]=0
r=2	*m*[1,2]=15750	*m*[2,3]=2625	*m*[3,4]=750	*m*[4,5]=1000	
r=3	*m*[1,3]=7875	*m*[2,4]=4375	*m*[3,5]=2500		
r=4	*m*[1,4]=9375	*m*[2,5]=7125			
r=5	*m*[1,5]=11875				

$$m[2,5] = \min\{0+2500+35\times15\times20,\ 2625+1000+35\times5\times20,\ 4375+0+35\times10\times20\} = 7125$$



(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]

动态规划求解方法(9/10)

标记函数s[i, j]：

r=2	s[1,2]=1	s[2,3]=2	s[3,4]=3	s[4,5]=4
r=3	s[1,3]=1	s[2,4]=3	s[3,5]=3	
r=4	s[1,4]=3	s[2,5]=3		
r=5	s[1,5]=3			

解的追踪： $s[1,5]=3 \Rightarrow (A1\,A2\,A3)(A4\,A5)$

$s[1,3]=1 \Rightarrow A1(A2\,A3)$

输出

计算顺序： $(A1(A2\,A3))(A4\,A5)$

最少的乘法次数： $m[1,5]=11875$

动态规划求解方法(10/10)

两种实现的比较：

递归实现：时间复杂性高，空间较小

动态规划：**时间复杂性低，空间消耗多**

原因：递归实现子问题多次重复计算，子问题计算次数呈指数增长. 迭代实现每个子问题只计算一次。

动态规划时间复杂度：

备忘录各项计算量之和 + 追踪解工作量

通常追踪工作量不超过计算工作量 ，是问题规模的多项式函数。

§3.2 动态规划算法的基本要素

1. 最优子结构性质

当问题的最优解包含了其子问题的最优解时，称为该问题具有最优子结构性质。

2. 重叠子问题性质

在用递归算法自顶向下解此问题时，每次产生的子问题并不总是新问题。有些子问题被反复计算多次。

动态规划正是利用子问题的重叠性质，对每一个子问题只解一次，而后保留起来，下次再用只需查看结果。

见P54

动态规划算法的基本要素

3、备忘录方法

- 备忘录方法的控制结构与直接递归方法的控制结构相同，区别在于备忘录方法为每个解过的子问题建立了备忘录以备需要时查看，避免了相同子问题的重复求解。

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

例如：合并果子-1

在一个果园里，果农已经将所有的果子打了下来，而且按圆形区域堆放了若干堆。最后果农要把所有的果子合并成一堆。

每一次合并，果农总是把两堆相邻果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过n-1次合并之后，就只剩下一堆了。果农在合并果子时总共消耗的体力等于每次合并所耗体力之和。

假定每个果子重量都为1，并且已知果子堆数和每堆的数目，你的任务是设计出合并的次序方案，使得果农耗费的体力最少，并输出这个最小的体力耗费值。

例如：N=6　3　4　6　5　4　2

一、考虑每次选相邻的最小两堆合并

3　4　6　5　4　2　→　5

5　4　6　5　4　→　9

9　6　5　4　→　9

9　6　9　→　15

15　9　→　24

24

62

例如：N=6 3 4 6 5 4 2

二、更好方案

3 4 6 5 4 2 → 7

7 6 5 4 2 → 13

13 5 4 2 → 6

13 5 6 → 11

13 11 → 24

24

$\boxed{61}$

利用动态规划求解1

相邻堆进行合并有多个子序列：

$\{a_1\ \ a_2\}$ $\{a_2\ \ a_3\}$ … $\{a_n\ \ a_1\}$

$\{a_1\ \ a_2\ \ a_3\}$ $\{a_2\ \ a_3\ \ a_4\}$ $\{a_n\ \ a_1\ \ a_2\}$

…

$\{a_1\ \ a_{2}\ldots\ a_{n-1}\}$….. $\{a_n\ \ a_{1}\ldots\ a_{n-2}\}$

利用动态规划求解2

为了便于运算，用[i，j]表示从第i堆起,顺时针的j堆组成的子序列。

Data[i][j]=a[i]+a[i+1]+…+a[i+j-1]

Fm[i][j] 表示从第i堆起,顺时针的j堆合并成一堆的最小值

显然

注意下标越界问题

Fm[i][1]=0;

Fm[i][j]=MIN{ Fm[i][k] + Fm[i+k][j-k] +data[i][j] }

k=2, 3, …, j-1

求Fm[1][n]

§3.3　　最长公共子序列

定义 **一个给定序列的子序列是在该序列中删除若干元素后得到的序列。**即

若给定序列 X={x1, x2, …, xm}，则另一序列 Z={z1, …, zk}是X的子序列是指：

存在一个严格递增下标序列{i1, i2, …, ik}使得对于所有j=1, 2, …, k有：$Z_j = X_{ij}$

例如　序列　　X={A，B，C，B，D，A，B}

子序列　　Z={B，C，D，B}

相应的递增下标序列为{2，3，5，7}。

公共子序列

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。

最长公共子序列问题

给定2个序列 X={x1, x2, ..., xm}和 Y={y1, y2, ..., yn}，找出 X 和 Y 的最长公共子序列。

例如：字符串13455和245576的最长公共子序列为455

字符串acdfg和adfc的最长公共子序列为adf

LCS（Longest Common Subsequence）的应用

- 求两个序列中最长的公共子序列算法，广泛的应用在图形相似出路、媒体流的相似比较、计算生物学方面。生物学家常常利用该算法进行基因序列比对，由此推测序列的结构、功能和演化过程。
- LCS可以描述两段文字之间的“相似度”，即它们的雷同程度，从而能够用来辨别抄袭。另一方面，对一段文字进行修改之后，计算改动前后文字的最长公共子序列，将除此子序列外的部分提取出来，这种方法判断修改的部分，往往十分准确。简而言之，百度知道，百度百科都用得上。

最长公共子序列的结构

求最长公共子序列问题，最容易想到的算法----枚举法

即，对X的所有子序列，检查它是否也是Y的子序列，从中找出最长的子序列。

但 X 共有2^M个不同的子序列。枚举法 $O(2^m)$

事实上，

最长公共子序列问题，具有最优子结构性质。

最优子结构性质。

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

- 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。
- 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
- 若$x_m \neq y_n$ 且 $z_k \neq y_n$ 则 Z是X_m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。

子问题的递归结构

由最长公共子序列问题的最优子结构性质建立子问题最优值的递归关系。

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，

$X_i=\{x_1, x_2, \ldots, x_i\}$　　　$Y_j=\{y_1, y_2, \ldots, y_j\}$

由最优子结构性质可建立递归关系如下：

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

计算最优值

由于在所考虑的子问题空间中，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的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]="↖ ";  }
        else
          if (c[i-1][j]>=c[i][j-1])    { c[i][j]=c[i-1][j];      b[i][j]="↑";   }
          else
                                       { c[i][j]=c[i][j-1];      b[i][j]="←";   }
}     }
```

O(mn)

构造最长公共子序列

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

O(m+n)

结果

i		j: 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

LCS:<B,C,B,A>

算法的改进

在算法LCSLength和LCS中，可进一步将数组b省去。

事实上，数组元素c[i][j]的值仅由c[i-1][j-1]，c[i-1][j]和c[i][j-1]这3个数组元素的值所确定。

如果只需要计算最长公共子序列的长度，则算法的空间需求可大大减少。事实上，在计算c[i][j]时，只用到数组c的第i行和第i-1行。因此，用2行的数组空间就可以计算出最长公共子序列的长度。进一步的分析还可将空间需求减至O(min(m,n))。

Question?

•题目：给定一个长度为N的数组，找出一个最长的单调递增子序列。例如：给定数组｛5，6，7，1，2，8｝则其最长的单调递增自序列为｛5，6，7，8｝，长度为4。

•求解：怎么用LCS解决这个问题？

•分析：原数组=｛5，6，7，1，2，8｝

排序后=｛1，2，5，6，7，8｝

§3.4 最大子段和

给定由N个整数（可能有负整数）组成的序列$a_1, a_2, \ldots, a_n$，求该序列形如 $a_i+a_{i+1}+\ldots+a_j$的子段和的最大值。

当所有整数均为负整数时，定义其最大子段和为0

例如：

当$\{a_1, a_2, \ldots, a_6\}=\{-1, 11, -4, 13, -5, -2\}$时

其最大子段和为 20

算法

算法1： 对所有的(i, j)对，顺序求和 $a_i+...+a_j$ 并比较出最大的和

算法2： 分治策略，将数组分成左右两半，分别计算左边的最大和、右边的最大和、跨边界的最大和，然后比较其中最大者

算法3： 动态规划

算法1 最大子段和问题的简单算法

思路如下：

以a_0开始:

$\{a_0\}$, $\{a_0, a_1\}$, $\{a_0, a_1, a_2\}$……$\{a_0, a_1, \cdots\cdots a_n\}$

共n个

以a_1开始:

$\{a_1\}$, $\{a_1, a_2\}$, $\{a_1, a_2, a_3\}$……$\{a_1, a_2, \cdots\cdots a_n\}$

共n-1个

……

以an开始:{an}共1个

一共(n+1)*n/2个连续子段，使用枚举，

那么应该可以得到以下算法：

算法1 最大子段和问题的简单算法

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

$O(n^3)$

改进后算法

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

$O(n^2)$

算法2 最大子段和问题的分治算法

从问题的解的结构可以看出，它适合分治法

将序列 a[1:n] 分为长度相等的两段 a[1: n/2] a[n/2+1: n] 分别求出这两段的最大子段和，则a[1:n] 的最大子段和有三种情形

（1） a[1:n] 的最大子段和与a[1: n/2]的最大子段和相同。

(在前半部分)

（2） a[1:n] 的最大子段和与 a[n/2+1: n]的最大子段和相同。

(在后半部分)

（3） a[1:n] 的最大子段和为 $a_i+a_{i+1}+\ldots+a_j$ (在中间部分)

其中：$1 \leq i \leq n/2$ $n/2+1 \leq j \leq n$

情形（1） 和（2） 可递归求得

情形（3）一定 包括元素 a[n/2] 和 a[n/2+1]

因此，

在a[1: n/2]中， 求s1=**max** $\{a_i+a_{i+1}+\ldots+a_{n/2}\}$ i=1,2,…..,n/2

在a[n/2+1: n]中 求s2=**max** $\{a_{n/2+1}+\ldots+a_j\}$ j=n/2+1, ……,n

则**s1+s2** 即为情形（3)时的最大值。

$$T(n)=\begin{cases} O(1) & n<=0 \\ 2T(n/2)+O(n) & n>0 \end{cases} \Rightarrow T(n)=O(n\log n)$$

求最大子段和问题分治算法1

```
int  MaxSubSum(int *a,  int L, int R )
{    int  sum=0;
    if (L==R) sum=(a[L]>0)?a[L]:0;
    else {      int C=(L+R)/2;
                int Lsum= MaxSubSum(a, L, C );
                int Rsum= MaxSubSum(a, C+1, R );
                int s1=0,   Lefts=0;
                for( i=C;  i>L;  - -i )
                {  lefts+=a[i];
                    if(lefts>s1) s1=lefts;
                }
```

求最大子段和问题分治算法2

```
int s2=0,   rights=0;
 for( j=C+1;  j<R;  ++j )
     {   rights +=a[j];
          if(rights >s2)  s2= rights; }
 sum=s1+s2;
 if (sum<Lsum)  sum=Lsum;
 if (sum<Rsum)  sum=Rsum;
}
return sum;
}
```

O(nlogn)

算法3　最大子段和问题的动态规划算法

从对上述分治算法的分析可看出

若记

$$b[j] = \text{MAX}\{\ a[i]+a[i+1]+\ldots\ldots+a[j]\ \} \quad i=1, 2,\ldots, j$$

则最大子段和为

$$\text{MAX}\{\ b[1],\ b[2],\ b[3],\ldots\ldots,\ b[n]\ \}$$

如何求b[j]?

$$b[j]=\begin{cases} b[j-1]+a[j] & \text{当 } b[j-1]>0 \text{ 时} \\ a[j] & \text{当 } b[j-1]<0 \text{ 时} \end{cases}$$

$j=1, 2,\ldots, n$

求最大子段和问题动态规划算法

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

O(n)