



中国科学技术大学 计算机科学与技术系
University of Science and Technology of China
DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

算法基础

Foundation of Algorithms

主讲人 徐云

Fall 2018, USTC



Part 1 Foundation

Part 2 Sorting and Order Statistics

Part 3 Data Structure

Part 4 Advanced Design and Analysis Techniques

chap 15 Dynamic Programming

chap 16 Greedy Algorithms

chap 17 Amortized Analysis

Part 5 Advanced Data Structures

Part 6 Graph Algorithms

Part 7 Selected Topics

Part 8 Supplement



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.1 方法概述

- 历史及研究问题
- 一些术语和概念
- 最优性原理
- 方法的基本思想
- 方法的求解步骤
- 动态规划法的适用条件
- 最优性原理判别举例
- 设计技巧——阶段划分和状态表示
- 存在的问题

历史及研究问题 (1)

- **动态规划(dynamic programming)**是运筹学的一个分支，20世纪50年代初美国数学家**R.E.Bellman**等人在研究**多阶段决策过程(multistep decision process)**的**优化问题**时，提出了著名的**最优化原理(principle of optimality)**，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——**动态规划**。
- **多阶段决策问题**：求解的问题可以划分为一系列相互联系的阶段，在每个阶段都需要作出决策，且一个阶段决策的选择会影响下一个阶段的决策，从而影响整个过程的活动路线，求解的目标是选择各个阶段的决策使整个过程达到最优。

历史及研究问题 (2)

- **动态规划**主要用于求解以时间划分阶段的动态过程的优化问题，但是一些与时间无关的静态规划(如线性规划、非线性规划)，**可以人为地引进时间因素**，把它视为多阶段决策过程，也可以用动态规划方法方便地求解。
- **动态规划**是考察问题的一种途径，或是求解某类问题的一种方法。
- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了**广泛的应用**。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便。

一些术语和概念

- **阶段**：把所给的问题的求解过程恰当地划分为若干个相互联系阶段。
- **状态**：状态表示每个阶段开始时，问题或系统所处的客观状况。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。
 - **状态的无后效性**：如果某阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有马尔科夫性。

注：适于动态规划法求解的问题具有状态的无后效性
- **策略**：各个阶段决策的确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为**子过程**，其对应的某个策略称为**子策略**。

最优性原理

- Bellman的原定义如下：

An optimal policy has the property that **whatever the initial state and initial decision are, then remaining decisions must constitute an optimal policy** with regard to the state resulting from first decision.

- Bellman最优性原理：

求解问题的一个最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。

注：对具有最优性原理性质的问题而言，如果有一决策序列包含有非最优的决策子序列，则该决策序列一定不是最优的。

方法的基本思想

- 动态规划的思想实质是分治思想和解决冗余。
- 与分治法类似的是
将原问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。
- 与分治法不同的是
经分解的子问题往往不是互相独立的。若用分治法来解，有些共同部分（子问题或子子问题）被重复计算了很多次。
- 如果能够保存已解决的子问题的答案，在需要时再查找，这样就可以避免重复计算、节省时间。动态规划法用一个表来记录所有已解的子问题的答案。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表方式。

方法的求解步骤

- ①找出最优解的性质，并刻画其最优子结构特征；
- ②递归地定义最优值（写出动态规划方程）；
- ③以自底向上的方式计算出最优值；
- ④根据计算最优值时记录的信息，构造最优解。

注：

- 步骤①~③是动态规划算法的基本步骤。如果对于只需要求出最优值的情形，步骤④可以省略；
- 若需要求出问题的一个最优解，则必须执行步骤④，步骤③中记录的信息是构造最优解的基础。

适用条件

动态规划法的有效性依赖于问题本身所具有的两个重要的适用性质

- 最优子结构

如果问题的最优解是由其子问题的最优解来构造，则称该问题具有最优子结构性质。

- 重叠子问题

在用递归算法自顶向下解问题时，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。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只解一次，而后将其解保存在一个表格中，在以后该子问题的求解时直接查表。

最优性原理判别举例 (1)

- 例1: 设 G 是一个有向加权图, 则 G 从顶点 i 到顶点 j 之间的最短路径问题满足最优性原理。

证明: (反证)

设 $i \sim i_p \sim i_q \sim j$ 是一条最短路径, 但其中子路径 $i_p \sim i_q \sim j$ 不是最优的,

假设最优的路径为 $i_p \sim i'_q \sim j$

则我们重新构造一条路径: $i \sim i_p \sim i'_q \sim j$

显然该路径长度小于 $i \sim i_p \sim i_q \sim j$, 与 $i \sim i_p \sim i_q \sim j$ 是顶点 i 到顶点 j 的最短路径相矛盾。

所以, 原问题满足最优性原理。 □

最优性原理判别举例 (2)

- 例2: 0-1背包问题Knap(1,n,c)满足最优性原理

$$\begin{aligned} & \max \sum_{i=l}^n v_i x_i \\ & \begin{cases} \sum_{i=l}^n w_i x_i \leq c \\ x_i \in \{0,1\} \quad l \leq i \leq n \end{cases} \end{aligned} \quad \overset{\text{记}}{\Rightarrow} \quad \text{Knap}(l, n, c)$$

证明: 设 (y_1, y_2, \dots, y_n) 是Knap(1,n,c)的一个最优解, 下证 (y_2, \dots, y_n) 是Knap(2,n,c- $w_1 y_1$)子问题的一个最优解。

若不然, 设 (z_2, \dots, z_n) 是Knap(2,n,c- $w_1 y_1$)的最优解, 因此有

$$\sum_{i=2}^n v_i z_i > \sum_{i=2}^n v_i y_i \quad \text{且} \quad \sum_{i=2}^n w_i z_i \leq c - w_1 y_1$$

$$\Rightarrow \quad v_1 y_1 + \sum_{i=2}^n v_i z_i > \sum_{i=1}^n v_i y_i \quad \text{又有} \quad w_1 y_1 + \sum_{i=2}^n w_i z_i \leq c$$

说明 (y_1, z_2, \dots, z_n) 是Knap(1,n,c)的一个更优解, 矛盾。 \square

最优性原理判别举例 (3)

- 例3: 最长路径问题不满足最优性原理。

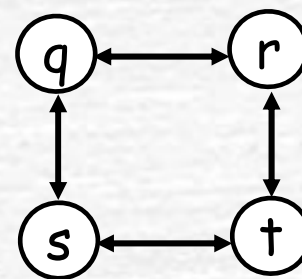
证明:

p: $q \rightarrow r \rightarrow t$ 是 q 到 t 的最长路径,

而 $q \rightarrow r$ 的最长路径是 $q \rightarrow s \rightarrow t \rightarrow r$

$r \rightarrow t$ 的最长路径是 $r \rightarrow q \rightarrow s \rightarrow t$

但 $q \rightarrow r$ 和 $r \rightarrow t$ 的最长路径合起来并不是 q 到 t 的最长路径。所以, 原问题并不满足最优性原理。 \square



注: 因为 $q \rightarrow r$ 和 $r \rightarrow t$ 的子问题都共享路径 $s \rightarrow t$, 组合成原问题解时, 有重复的路径对原问题是不允许的。

设计技巧

- 动态规划的设计技巧：**阶段的划分、状态的表示和存储表的设计**；
- 在动态规划的设计过程中，**阶段的划分和状态的表示是其中重要的两步**，这两步会直接影响该问题的计算复杂性和存储表设计，有时候阶段划分或状态表示的不合理还会使得动态规划法不适用。
- 记忆型递归——动态规划的变种
 - 每个子问题的解对应一表项；
 - 每表项初值为一特殊值，表示尚未填入；
 - 递归时，第一次遇到子问题进行计算并填表，以后查表取值；如：矩阵链乘的记忆型递归算法P220

存在的问题

- 问题的阶段划分和状态表示，需要具体问题具体分析，没有一个清晰明朗的方法；
- 空间溢出的问题，是动态规划解决问题时一个普遍遇到的问题；
 - 动态规划需要很大的空间以存储中间产生的结果，这样可以使包含同一个子问题的所有问题共用一个子问题解，从而体现动态规划的优越性，但这是以牺牲空间为代价的，为了有效地访问已有结果，数据也不易压缩存储，因而空间矛盾是比较突出的。



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.2 多段图规划

- 问题描述及举例
- 问题满足最优性原理
- 递归关系推导
- 算法
- 时间分析

问题描述及举例 (1)

- 问题描述

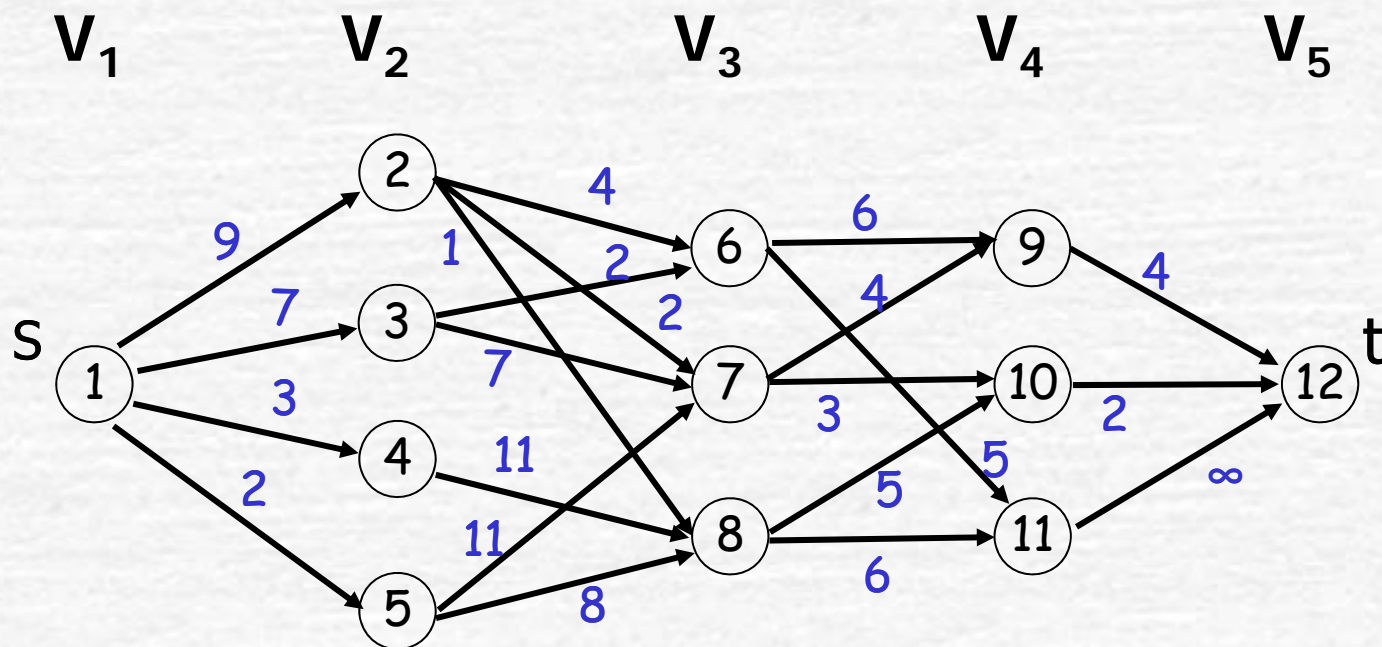
多段图 $G=(V, E)$ 是一个有向图，且具有以下特征：

- (1) 划为 $k \geq 2$ 个不相交的集合 V_i , $1 \leq i \leq k$;
- (2) V_1 和 V_k 分别只有一个结点 s (源点)和 t (汇点);
- (3) 若 $\langle u, v \rangle \in E(G)$, $u \in V_i$, 则 $v \in V_{i+1}$ $1 \leq i \leq k$, 边上成本记 $c(u, v)$; 若 $\langle u, v \rangle \in E(G)$, 边上成本记 $c(u, v) = \infty$;

求由 s 到 t 的最小成本路径。

问题描述及举例 (2)

- 举例：一个5-段图



求一条由 s 到 t 的成本最小的路径？

最优性原理和递归式

- 多段图问题满足最优性原理

设 $s, \dots, v_{i_p}, \dots, v_{i_q}, \dots, t$ 是一条由 s 到 t 的最短路径，
则 $v_{i_p}, \dots, v_{i_q}, \dots, t$ 也是由 v_{i_p} 到 t 的最短路径。（反证即可）

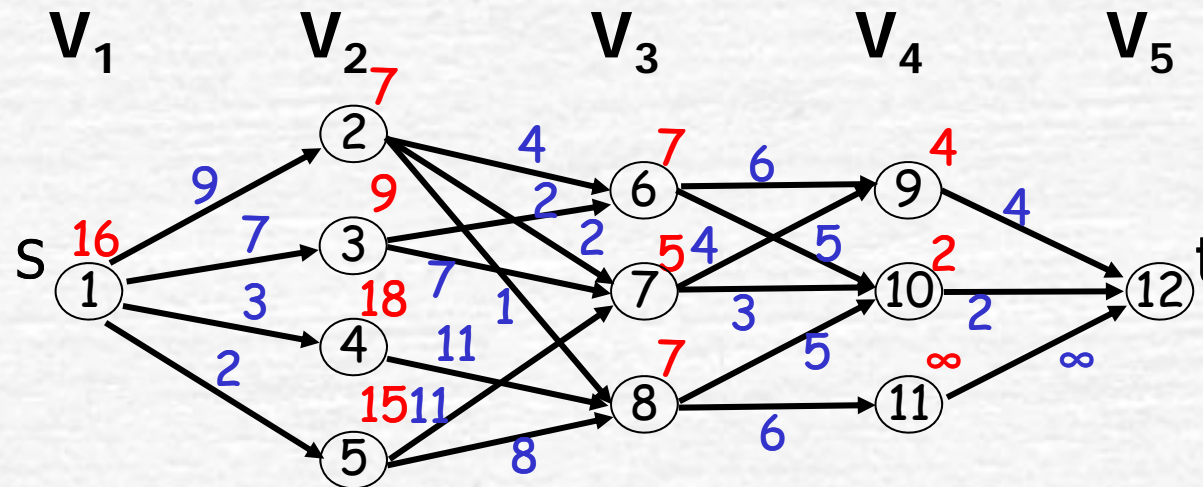
- 递归式推导

设 $\text{cost}(i, j)$ 是 V_i 中结点 v_j 到汇点 t 的最小成本路径的成本，递归式为：

$$\text{cost}(i, j) = \begin{cases} c(j, t) & i = k - 1 \\ \min_{\substack{v_l \in V_{i+1} \\ \langle j, l \rangle \in E(G)}} \{c(j, l) + \text{cost}(i + 1, l)\} & 1 \leq i < k - 1 \end{cases}$$

多段图规划算法 (1)

- 计算过程(以5-段图为例)



$$\text{cost}(4,9)=4, \text{cost}(4,10)=2, \text{cost}(4,11)=\infty$$

$$\text{cost}(3,6)=7, \text{cost}(3,7)=5, \text{cost}(3,8)=7$$

$$\text{cost}(2,2)=7, \text{cost}(2,3)=9, \text{cost}(2,4)=18, \text{cost}(2,5)=15$$

$$\text{cost}(1,1)=\min\{9+\text{cost}(2,2), 7+\text{cost}(2,3), 3+\text{cost}(2,4), 2+\text{cost}(2,5)\}=16$$

构造解: 解1(1,2,7,10,12), 解2(1,3,6,10,12)

多段图规划算法 (2)

MultiStageGraph($G, k, n, p[]$)

{//输入 n 个结点的 k 段图, 假设顶点按段的顺序编号

// $E(G)$ 是边集, $p[1..k]$ 是最小成本路径

new cost[n]; //生成数组cost, cost[j]相当于前面的cost(i,j)

new d[n]; //生成数组d, d[j]保存 v_j 与下一阶段的最优连接点

cost[n]=0;

for i=n-1 downto 1 do //计算cost[i]和d[i]

{ cost[i]= ∞ ;

while(任意 $\langle i, r \rangle \in E(G)$) //r是下一阶段中的顶点

if($c(i, r) + \text{cost}[r] < \text{cost}[i]$)

{ cost[i]= $c(i, r) + \text{cost}[r]$; d[i]=r;

}

}

p[1]=1; p[k]=n; //以下是找一条最小成本路径 (构造解)

for i=2 to k-1 do p[i]=d[p[i-1]];

}

$\therefore T(n) = O(n+e)$

$O(k)$

$O(n+e)$



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.3 矩阵链乘法

- 问题描述
- 加括号的方案数
- 动态规划算法

问题描述

- 问题描述:

给定 n 个矩阵 A_1, A_2, \dots, A_n , A_i 的维数为 $p_{i-1} \times p_i (1 \leq i \leq n)$, 以一种最小化标量乘法次数的方式进行完全括号化。

- Remark:

1. 设 $A_{p \times q}, A_{q \times r}$ 两矩阵相乘, 普通乘法的次数为 $p \times q \times r$

2. 加括号对乘法次数的影响

如: $A_{10 \times 100} \times B_{100 \times 5} \times C_{5 \times 50}$

$((AB)C)$: 7500次

$(A(BC))$: 75000次

加括号的方案数

用 $p(n)$ 表示 n 个矩阵链乘的穷举法计算成本，如果将 n 个矩阵从第 k 和第 $k+1$ 处隔开，对两个子序列再分别加括号，用 $p(n)$ 表示则可以得到下面递归式：

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

$\Rightarrow p(n) = C(n-1)$ 为Catalan数

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right) \quad \text{呈指数增长}$$

因此，穷举法不是一个有效算法

动态规划算法 (1)

- 最优性原理分析

1. 矩阵链乘问题满足最优性原理

记 $A[i:j]$ 为 $A_i A_{i+1} \dots A_j$ 链乘的一个最优括号方案，设 $A[i:j]$ 的最优次序中含有二个子链 $A[i:k]$ 和 $A[k+1:j]$ ，则 $A[i:k]$ 和 $A[k+1:j]$ 也是最优的。（反证可得）

2. 矩阵链乘的子问题空间： $A[i:j]$, $1 \leq i \leq j \leq n$

$A[1:1]$, $A[1:2]$, $A[1:3]$,	...	$A[1:n]$
$A[2:2]$, $A[2:3]$,	...	$A[2:n]$
...
	$A[n-1:n-1]$,	$A[n-1:n]$
		$A[n:n]$

动态规划算法 (2)

- 递归求解最优解的值

记 $m[i][j]$ 为计算 $A[i:j]$ 的最少乘法数，则原问题的最优值为 $m[1][n]$ ，那么有

$$m[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1}p_kp_j\} & i < j \end{cases}$$

这里，

$$(A_i A_{i+1} \dots A_k)_{p_{i-1} \times p_k} \times (A_{k+1} A_{k+2} \dots A_j)_{p_k \times p_j}$$

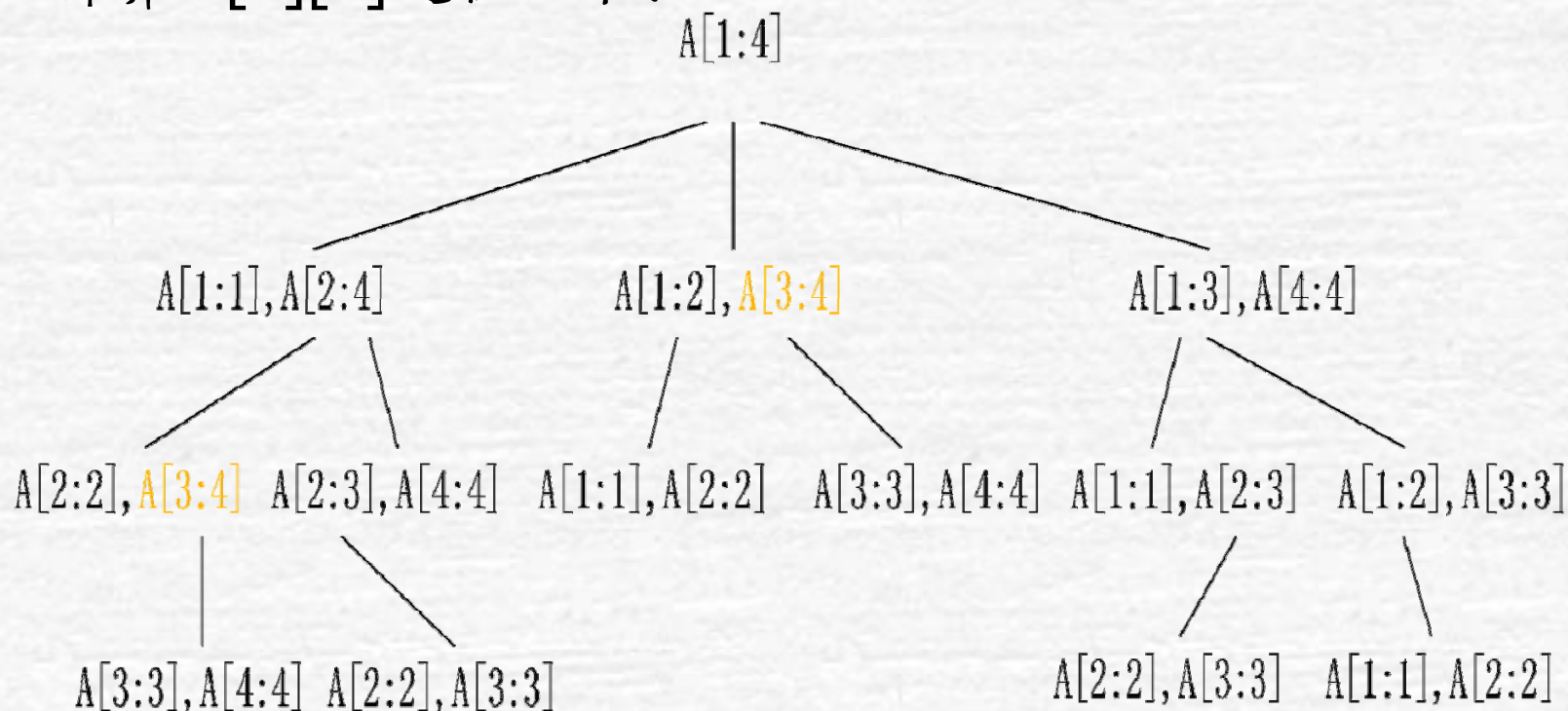
取得的 k 为 $A[i:j]$ 最优次序中的断开位置，并记录到表 $s[i][j]$ 中，即 $s[i][j] \leftarrow k$

注： $m[i][j]$ 实际是子问题最优解的解值，保存下来避免重复计算。

动态规划算法 (3)

- 递归求解最优解的值(Cont.)

计算 $m[1][4]$ 过程如下:

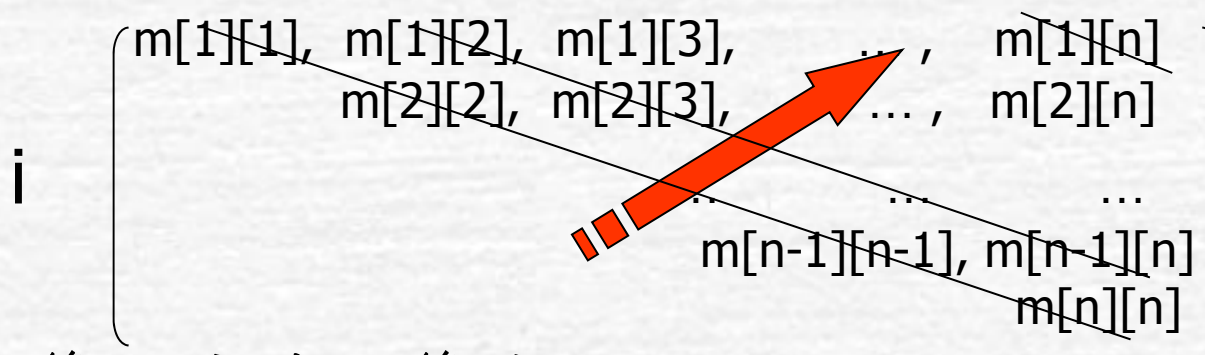


如 $A[3:4]$ 被计算了2次, 保存下来可以节省许多时间

动态规划算法 (4)

- 自底向上记忆化方式求解 $m[i][j]$

- 计算方向：以链长 l 递增方向. j



- 计算最优值的算法(Godbole,1973): P213

$$T(n)=O(n^3), S(n)=O(n^2)$$

- 注：①如果自顶向下计算(含重复计算)，这样效率较低(为 $\Omega(2^n)$)。

②也可以采用自顶向下的记忆型递归算法P220。

③Hu和Shing(1980,1982,1984)找到 $O(n \log n)$ 算法

动态规划算法 (5)

- 构造最优解

- 利用 $s[i][j]$ 中保存的 k , 进行对 $A[i:j]$ 的最佳划分, 加括号
为 $(A_i A_{i+1} \dots A_k) \times (A_{k+1} A_{k+2} \dots A_j)$
- 构造最优解的算法: P215

```
PrintOptimalParens(s, i, j)
{
    if i=j then
        print "A"i;
    else
        {
            print "(";
            PrintOptimalParens(s, i, s[i,j]);
            PrintOptimalParens(s, s[i,j]+1, j);
            print ")";
        }
}
```


动态规划算法 (6)

- 计算示例:

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

		j					
i	S	1	2	3	4	5	6
	1		1	1	3	3	3
	2			2	3	3	3
	3				3	3	3
	4					4	5
	5						5
	6						

		j					
i	m	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



Result

$((A_1(A_2A_3))((A_4A_5)A_6))$

$$m[2][5] = \min \begin{cases} m[2][2] + m[3][5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2][3] + m[4][5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125 \\ m[2][4] + m[5][5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.4 最大子段和

- 问题描述
- 直接算法: $T(n)=O(n^2)$
- 分治算法: $T(n)=O(n\log n)$
- 动态规划算法: $T(n)=O(n)$

问题描述

- 给定整数序列 a_1, a_2, \dots, a_n ，求形如 $\sum_{k=i}^j a_k$ 的子段和的最大值。规定子段和为负整数时，定义其最大子段和为0，即

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

- 例如， $(a_1, a_2, a_3, a_4, a_5, a_6) = (-2, 11, -4, 13, -5, -2)$
最大子段和为

$$\sum_{k=2}^4 a_k = 20$$

直接算法

MaxSubSum1(n, a[], besti, bestj)

{ //数组a[]存储a_i, 返回最大子段和, 保存起止位置到Besti,Bbestj中

sum=0;

for i=1 to n do

for j=i to n do

{ thissum=0;

for k=i to j do

//可以改进, 省略此循环

thissum += a[k];

if(thissum>sum)

{ sum=thissum;

besti=i; bestj=j;

}

}

return sum;

}

注: 原算法: $T(n)=O(n^3)$;

思考题: 对k循环可以省略, 改进后的算法: $T(n)=O(n^2)$;

分治算法 (1)

- 基本思想

将 $A[1..n]$ 分为 $a[1..n/2]$ 和 $a[n/2+1..n]$ ，分别对两区段求最大子段和，这时有三种情形：

Case 1: $a[1..n]$ 的最大子段和的子段落在 $a[1..n/2]$;

Case 2: $a[1..n]$ 的最大子段和的子段落在 $a[n/2..n]$;

Case 3: $a[1..n]$ 的最大子段和的子段跨在 $a[1..n/2]$ 和 $a[n/2..n]$ 之间;

分治算法 (2)

- 基本思想(Cont.)

对Case 1和Case 2可递归求解；

对Case 3，可知 $a[n/2]$ 和 $a[n/2+1]$ 一定在最大和的子段中，因此

在 $a[1..n/2]$ 中计算：
$$S_1 = \max_{1 \leq i \leq n/2} \sum_{k=i}^{n/2} a_k$$

在 $a[n/2..n]$ 中计算：
$$S_2 = \max_{n/2+1 \leq i \leq n} \sum_{k=n/2+1}^i a_k$$

易知： $S_1 + S_2$ 是Case 3的最大值

分治算法 (3)

● 算法

MaxSubSum2(a[], left, right)

{ //返回最大子段和

sum=0;

if(left=right)

sum=a[left]>0?a[left]:0;

else

{ center=(left+right)/2;

leftsum=

MaxSubSum2(a, left,center);

rightsum=

MaxSubSum2(a, center+1, right);

s1=0; leftmidsum=0;

for i=center to left do

{ leftminsum += a[i];

if (leftmidsum>s1) then

s1=leftmidsum;

}

s2=0; rightmidsum=0;

for i=center+1 to right do

{ rightminsum += a[i];

if(rightmidsum>s2) then

s2=rightmidsum;

}

sum=s1+s2;

if(sum<leftsum) then sum=leftsum;

if(sum<rightsum) then sum=rightsum;

} //end if

return sum;

}//end

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

$$\Rightarrow T(n) = O(n \log n)$$

动态规划算法 (1)

- 基本思想

- 子问题定义

考虑所有下标以j结束的最大的子段和 $b[j]$, 即

$$b[j] = \max_{1 \leq i \leq j} \left\{ \max \left\{ 0, \sum_{k=i}^j a_k \right\} \right\} \quad j = 1, 2, \dots, n$$

- 原问题与子问题的关系

$$\max_{1 \leq i \leq j \leq n} \left\{ \max \left\{ 0, \sum_{k=i}^j a_k \right\} \right\} = \max_{1 \leq j \leq n} \left\{ \max_{1 \leq i \leq j} \left\{ \max \left\{ 0, \sum_{k=i}^j a_k \right\} \right\} \right\} = \max_{1 \leq j \leq n} \{b[j]\}$$

- 子问题解的递归关系

$$b[j] = \begin{cases} \max\{a_1, 0\} & j = 1 \\ \max\{b[j-1] + a_j, 0\} & j > 1 \end{cases}$$

动态规划算法 (2)

- 算法

```
int MaxSubSum3(int n, int a[])
{
    int sum=0, b=0; //sum存储当前最大的b[j], b存储b[j]
    for(int j=1; j<=n; j++)
    {
        b += a[j];
        if(b<0) then b=0; // b[j]
        if(b>sum) then sum=b;
    }
    return sum;
}
```

- 运行时间: $T(n)=O(n)$

- 思考题

如果要记录最大子段的区间，如何修改程序？



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.5 最长公共子序列 (LCS)

- 问题描述
- 如何求 X 、 Y 的LCS
 - LCS最优解结构特征 (step1)
 - 子问题的递归解 (step2)
 - 计算最优解值 (step3)
 - 构造一个LCS (step4)

问题描述 (1)

- 子序列定义

给定序列 $X=(x_1, x_2, \dots, x_m)$ ，序列 $Z=(z_1, z_2, \dots, z_k)$ 是 X 的一子序列，必须满足：若 X 的索引中存在一个严格增的序列 i_1, i_2, \dots, i_k ，使得对所有的 $j=1 \sim k$ ，均有 $x_{i_j}=z_j$

例如，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 $\{2, 3, 5, 7\}$ 。

- 两个序列的公共子序列

Z 是 X 和 Y 的子序列，则 Z 是两者的公共子序列 CS 。

- 最长的公共子序列(LCS)

在 X 和 Y 的 CS 中，长度最大者为一个最长公共子序列 LCS 。

问题描述 (2)

- Example:

In biological application, given two DNA sequences, for instance

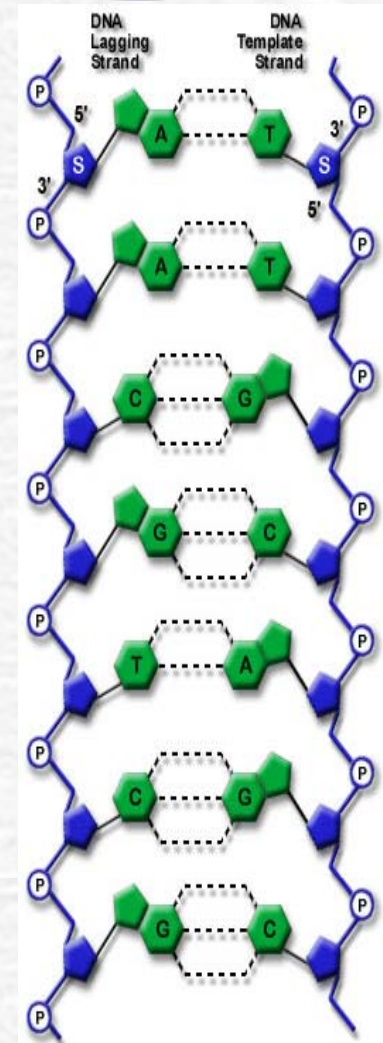
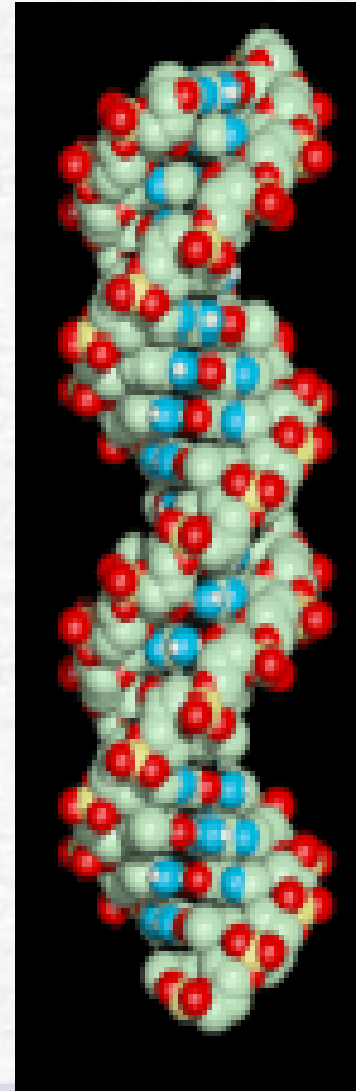
$S_1 =$
ACCGGTCGAGTGCGCGGAAGCCG
GCCGAA

and

$S_2 =$
GTCGTTCGGAATGCCGTTGCTCT
GTAAA,

how to compare them?

We have various standards of similarity for distinct purposes. While, the LCS of S_1 and S_2 is $S_3 =$ GTCGTCGGAAGCCGGCCGAA.



© 2001 Regents of N.M. State Univ./SWBIC

求LCS的step1 (1)

- Step1: LCS最优解的结构特征

定义 X 的 i^{th} 前缀: $X_i=(x_1, x_2, \dots, x_i)$, $i=1 \sim m$

$X_0=\varphi$ φ 为空集

- Th15.1 (一个LCS的最优子结构)

设序列 $X=(x_1, x_2, \dots, x_m)$ 和 $Y=(y_1, y_2, \dots, y_n)$, $Z=(z_1, z_2, \dots, z_k)$ 是 X 和 Y 的任意一个LCS, 则

(1) 若 $x_m=y_n$, $\implies z_k=x_m=y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS;

(2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$, $\implies Z$ 是 X_{m-1} 和 Y 的一个LCS;

(3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$, $\implies Z$ 是 X 和 Y_{n-1} 的一个LCS;

注: 由此可见, 2个序列的最长公共子序列可由(1)(2)(3)算出, (2)(3)的解是对应子问题的最优解。因此, 最长公共子序列问题具有最优子结构性质。

求LCS的step1 (2)

- Th15.1 的证明

(1) 若 $x_m = y_n$, $\implies z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS;
(应用反证法)

先证: $z_k = x_m = y_n$.

若 $z_k \neq x_m$ (也有 $z_k \neq y_n$), 则将 x_m 加到 Z 后, 于是获得 X 和 Y 的长度为 $k+1$ 的CS, 与 Z 是 X 和 Y 的LCS矛盾。

$\implies z_k = x_m = y_n$

再证: Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个LCS。

由 Z 的定义 \implies 前缀 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的CS (长度为 $k-1$)

若 Z_{k-1} 不是 X_{m-1} 和 Y_{n-1} 的LCS, 则存在一个 X_{m-1} 和 Y_{n-1} 的公共子序列 W , W 的长度 $> k-1$, 于是将 z_k 加入 W 之后, 则产生的公共子序列长度 $> k$, 与 Z 是 X 和 Y 的LCS矛盾。

$\implies Z_{k-1}$ 是 X_{m-1} 和 Y_{n-1} 的一个LCS

求LCS的step1 (3)

- Th15.1 的证明

(2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$, $\implies Z$ 是 X_{m-1} 和 Y 的一个 LCS;

$\because z_k \neq x_m$, 则 Z 是 X_{m-1} 和 Y 的一个 CS

下证: Z 是 X_{m-1} 和 Y 的 LCS

(反证) 若不然, 则存在长度 $> k$ 的 CS 序列 W ,

显然, W 也是 X 和 Y 的 CS, 但其长度 $> k$, 矛盾。

(3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$, $\implies Z$ 是 X 和 Y_{n-1} 的一个 LCS;

(3) 与 (2) 对称, 类似可证。

综上, 定理15.1证毕。

□

求LCS的step2

- Step2: 子问题的递归解

– 定理15.1将X和Y的LCS分解为:

(1) if $x_m = y_n$ then //解一个子问题

找 X_{m-1} 和 Y_{n-1} 的LCS;

(2) if $x_m \neq y_n$ then //解二个子问题

找 X_{m-1} 和Y的LCS 和 找X和 Y_{n-1} 的LCS;

取两者中的最大的;

– $c[i,j]$ 定义为 X_i 和 Y_j 的LCS长度, $i=0\sim m, j=0\sim n$;

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

求LCS的step3 (1)

- Step3: 计算最优解值

- 数据结构设计

$c[0..m, 0..n]$ //存放最优解值, 计算时行优先

$b[1..m, 1..n]$ //解矩阵, 存放构造最优解信息

$b[i, j] = \begin{cases} \swarrow & \text{如果 } c[i, j] \text{ 由 } c[i-1, j-1] \text{ 确定} \\ \uparrow & \text{如果 } c[i, j] \text{ 由 } c[i-1, j] \text{ 确定} \\ \leftarrow & \text{如果 } c[i, j] \text{ 由 } c[i, j-1] \text{ 确定} \end{cases}$

当构造解时, 从 $b[m, n]$ 出发, 上溯至 $i=0$ 或 $j=0$ 止

上溯过程中, 当 $b[i, j]$ 包含“ \swarrow ”时打印出 $x_i(y_j)$

求LCS的step3 (2)

- Step3: 计算最优解值

- 算法

LCS_Length(X, Y)

```
{  m ← length[X]; n ← length[Y];  
  for i ← 0 to m do c[i,0] ← 0; //0列  
  for j ← 0 to n do c[0,j] ← 0; //0行  
  for i ← 1 to m do  
    for j ← 1 to n do  
      if  $x_i = y_j$  then  
        { c[i, j] ← c[i-1, j-1] + 1; b[i, j] ← "↖"; }  
      else  
        if c[i-1, j] ≥ c[i, j-1] then  
          { c[i, j] ← c[i-1, j]; b[i, j] ← "↑"; } //由 $X_{i-1}$ 和 $Y_j$ 确定  
        else  
          { c[i, j] ← c[i, j-1]; b[i, j] ← "←"; } //由 $X_i$ 和 $Y_{j-1}$ 确定  
  return b and c;  
}
```

时间: $\theta(mn)$

求LCS的step3 (3)

Example

$i \backslash j$	0	1	2	3	4	5	6
y_j	0	B	D	C	A	B	A
0 x_i	0	0	0	0	0	0	0
1 A	0	\uparrow 0	\uparrow 0	\uparrow 0	\swarrow 1	\leftarrow 1	\swarrow 1
2 B	0	\swarrow 1	\leftarrow 1	\leftarrow 1	\uparrow 1	\swarrow 2	\leftarrow 2
3 C	0	\uparrow 1	\uparrow 1	\swarrow 2	\leftarrow 2	\uparrow 2	\uparrow 2
4 B	0	\swarrow 1	\uparrow 1	\uparrow 2	\uparrow 2	\swarrow 3	\leftarrow 3
5 D	0	\uparrow 1	\swarrow 2	\uparrow 2	\uparrow 2	\uparrow 3	\uparrow 3
6 A	0	\uparrow 1	\uparrow 2	\uparrow 2	\swarrow 3	\uparrow 3	\swarrow 4
7 B	0	\swarrow 1	\uparrow 2	\uparrow 2	\uparrow 3	\swarrow 4	\uparrow 4

求LCS的step4

- Step4: 构造一个LCS

- 算法

```
Print_LCS(b, X, i, j)
```

```
{ if i=0 or j=0 then return;
```

```
  if b[i,j]="↖" then
```

```
    { Print_LCS(b, X, i-1, j-1);
```

```
      print  $x_i$ ;
```

```
    }
```

```
  else
```

```
    if b[i,j]="↑" then Print_LCS(b, X, i-1, j);
```

```
    else Print_LCS(b, X, i, j-1);
```

```
}
```

时间 : $\Theta(m+n)$



第15章 动态规划

15.1 方法概述

15.2 多段图规划

15.3 矩阵链乘法

15.4 最大子段和

15.5 最长公共子序列

15.6 0-1背包

15.6 0-1 背包问题

- 问题描述及举例
- 问题满足最优性原理
- 递归关系推导
- 算法
- 时间分析

问题描述及举例

- 问题描述: $\text{Knap}(l, n, c)$

$\text{Knap}(l, n, c)$ 定义如下:

$$\max \sum_{i=l}^n v_i x_i \quad v_i > 0$$

$$\begin{cases} \sum_{i=l}^n w_i x_i \leq c & w_i > 0 \\ x_i \in \{0, 1\} & l \leq i \leq n \end{cases} \quad \text{求}(x_l, x_{l+1}, \dots, x_n) \text{使目标函数最大}$$

- 例如: $w=(w_1, w_2, w_3)=(2, 3, 4)$, $v=(v_1, v_2, v_3)=(2, 3, 4)$,
 $n=3$, $c=6$, 求 $\text{Knap}(1, 3, 6)$

取 $x=(1, 0, 1)$ 时, $\sum_{i=1}^n v_i x_i = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot 1 = 6$ 最大

用穷举法求解, 时间复杂度为 $O(n2^n)$

- 最优性原理证明: (见第1节)

递归关系

- 考虑子问题：子问题的背包容量在变化

$\text{Knap}(i, n, j)$ $j \leq c$ (假设 c, w_i 取整数)

设其最优值为 $m(i, j)$, 即 $m(i, j)$ 是背包容量为 j , 可选物品为 $i, i+1, \dots, n$ 的 0-1 背包问题的最优值。

- 递归式如下：不取物品 i

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

取物品 i

说明：当 $j < w_i$ 时，只有 $x_i = 0$, $\therefore m(i, j) = m(i+1, j)$;

$$\text{当 } j \geq w_i \text{ 时, } \begin{cases} \text{取 } x_i = 0 \text{ 时, 为 } m(i+1, j) \\ \text{取 } x_i = 1 \text{ 时, 为 } m(i+1, j-w_i) + v_i \end{cases}$$

$$\text{临界条件: } m(n, j) = \begin{cases} v_n & j \geq w_n \\ 0 & 0 \leq j < w_n \end{cases} \quad (3)$$

(4)

0-1背包问题算法 (1)

Knapsack(v[], w[], c, n, m[][])

{//输出m[1][c]

jMax=min(w[n]-1, c); //j≤jMax, 即 $0 \leq j < w_n$; j>jMax, 即 $j \geq w_n$

for j=0 to jMax do m[n][j]=0; //0≤j<w_n, (4)式

for j=w[n] to c do m[n][j]=v[n]; //j≥w_n, (3)式

for i=n-1 downto 2 do //i>1表示对i=1暂不处理, i=1时只需求m[1][c]

{ jMax=min(w[i]-1, c);

for j=0 to jMax do //0≤j<w_i, (2)式

m[i][j]=m[i+1][j];

for j=w[i] to c do //j≥w_i, (1)式

m[i][j]=max(m[i+1][j], m[i+1][j-w[i]]+v[i]);

}

if c≥w[1] then m[1][c]=max(m[2][c], m[2][c-w[1]]+v[1]);

else m[1][c]=m[2][c];

}

0-1背包问题算法 (2)

```
Traceback( w[], c, n, m[][], x[])
{ // 输出解 x[1..n]
  for i=0 to n do
    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;
}
```

- 运行时间: $T(n)=O(cn)$

注: 当 $c > 2^n$ 时, 算法仍需 $O(n2^n)$



End of Ch15

