西南大學

动态规划算法

张里博

lbzhang@swu.edu.cn

目录

PART 01

动态规划算法的设计思想

Dynamic Programming

动态规划算法

动态规划(dynamic programming)算法是一种求解**多阶段决策问题**的算法设计技术。动态规划是运筹学的一个分支，也是求解决策过程(decision process)最优化问题的数学方法。

主要思想：将原问题归约为**规模较小、结构相同**的子问题，建立原问题与子问题**优化函数间（最优解或最优目标函数值）的依赖关系；**

从规模最小的子问题开始，利用上述依赖关系求解规模**更大的子问题**，直至得到原问题的解。

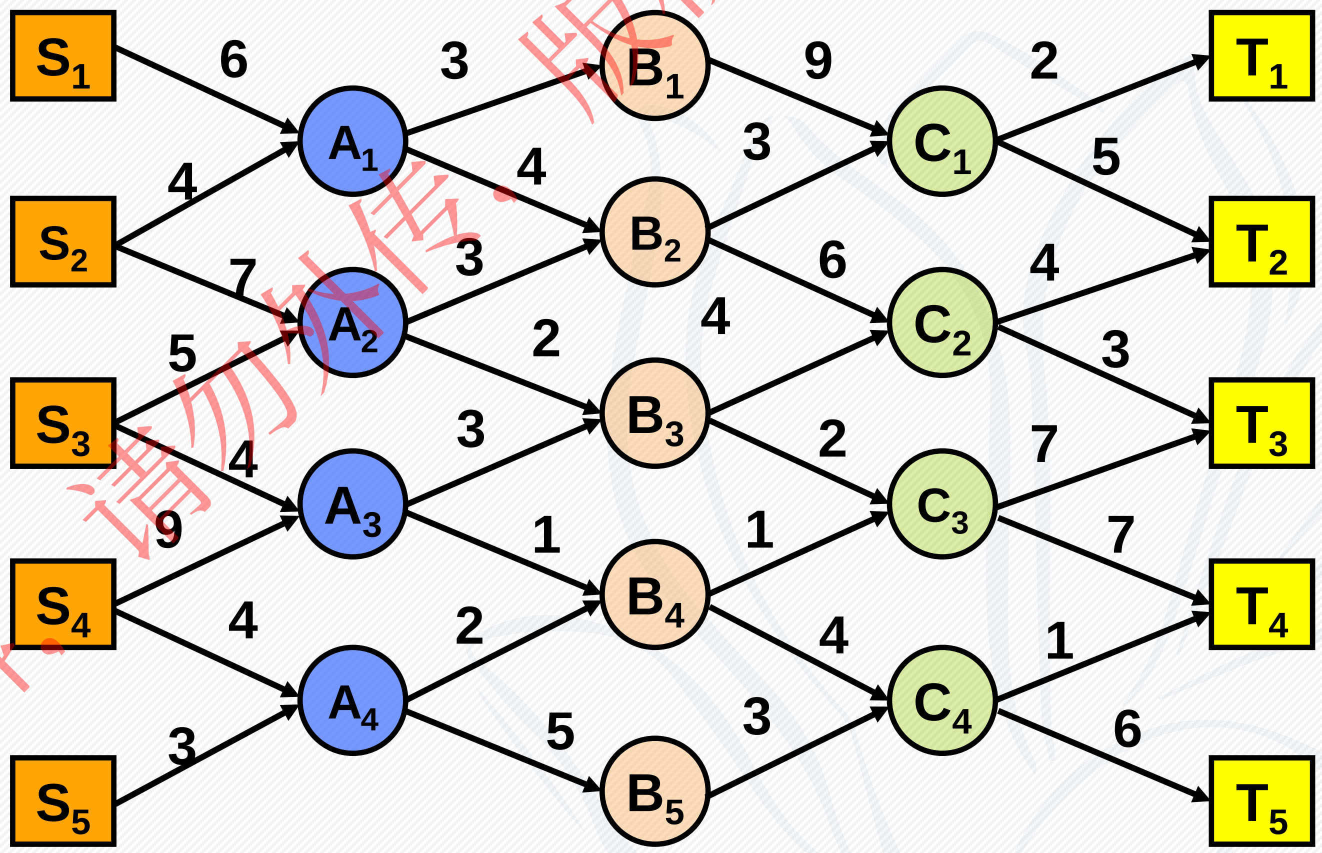
问题描述

输入：

- 起点集合$\{S_1, S_2, ..., S_m\}$，
- 终点集合$\{T_1, T_2, ..., T_m\}$，
- n层有向图，
- 边集E，其中任意边e有长度；

输出：

一条从（任意）起点到（任意）终点的最短路径（路径总长度最短）



蛮力算法

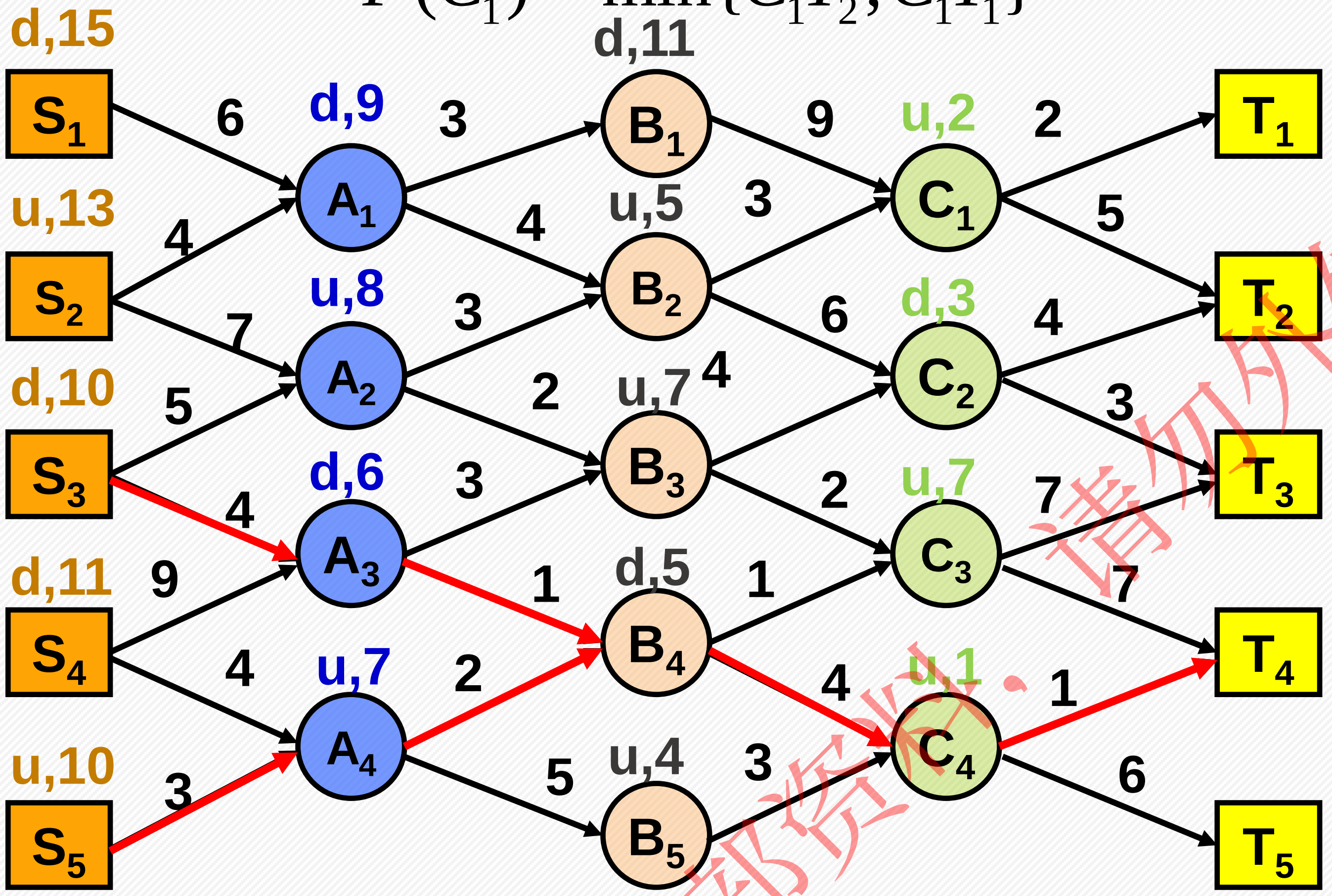
- 计算每一条所有从起点到终点的路径，计算长度，从其中找出最短路径。

- 在上述实例中，如果网络的层数为n，起点个数为m个，那么每个起点到所有终点路径条数将接近于2^{n-1}（每个结点最多有2个分支与下一层相连），总时间复杂度为$O(m2^n)$

- 动态规划算法：
- 从终点出发，起点逐步前移，每一步计算当前层的节点与终点的最短路径及长度。将**多阶段决策过程转化成多个单阶段决策过程**。

最短路径问题

$$F(C_1)=\min\{C_1T_2, C_1T_1\}$$

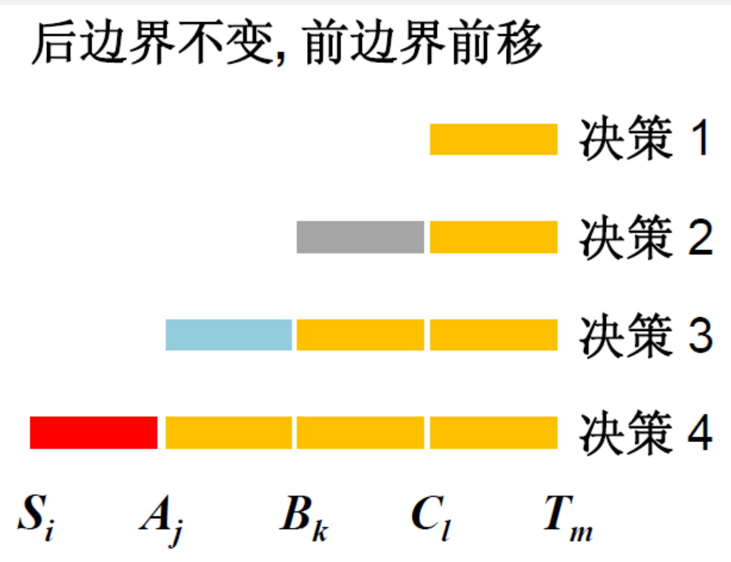


$$F(B_2)=\min\{B_2C_1+F(C_1), B_2C_2+F(C_2)\}$$

$$=\min\{\min\{B_2C_1+C_1T_2, B_2C_1+C_1T_1\}, \min\{B_2C_2+C_2T_2, B_2C_2+C_2T_3\}\}$$

子问题的界定

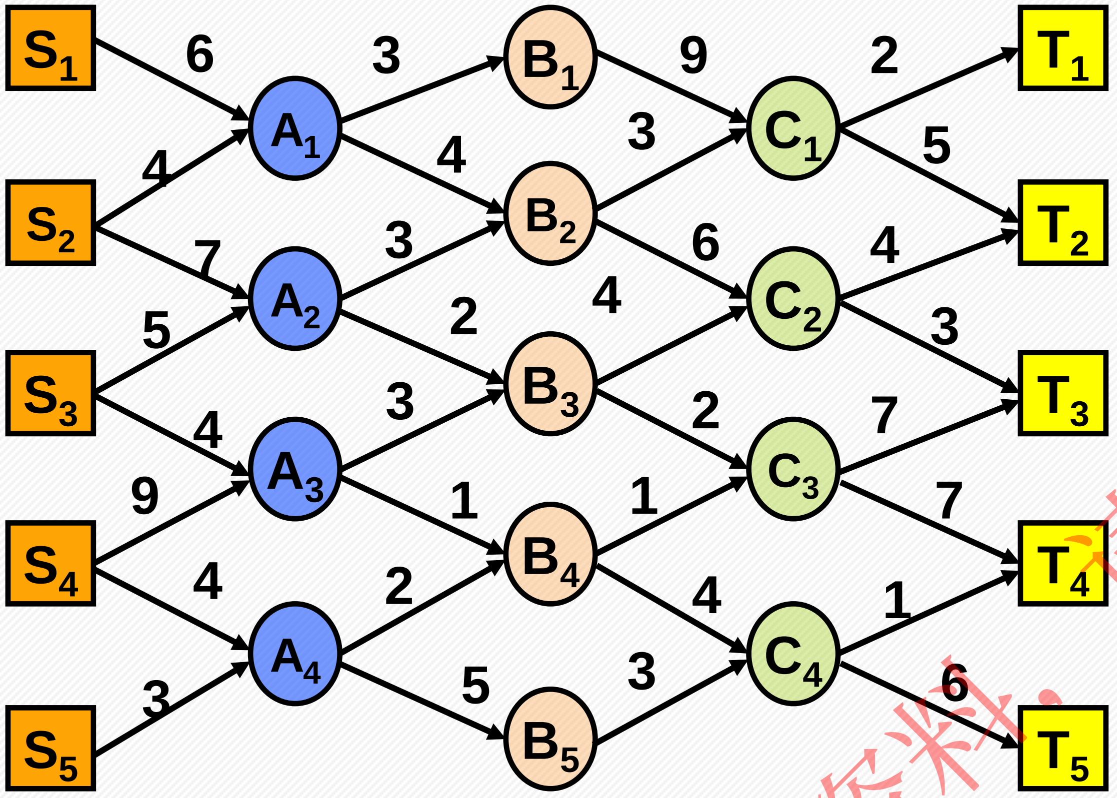
从终点层出发，起点逐步前移，每步计算当前层节点与终点的最短路径及长度



每步决策将依赖于以前步骤的决策结果。

最短路径问题

$$F(B_2)=\min\{B_2C_1+F(C_1),B_2C_2+F(C_2)\}$$



动态规划只考虑当前子问题的最优解延伸的结果，许多不可能最优的路径已被删除。

最短路径长度的依赖关系

决策一 $F(C_l)=\min\limits_{m}\{C_lT_m\}$

决策二 $F(B_k)=\min\limits_{l}\{B_kC_l+F(C_l)\}$

决策三 $F(A_j)=\min\limits_{k}\{A_jB_k+F(B_k)\}$

决策四 $F(S_i)=\min\limits_{j}\{S_iA_j+F(A_j)\}$

时间复杂度 O(3*m*(n-1))=O(mn)

每步求解的问题是后面阶段求解问题的子问题。

问题特点

- 例3.1动态规划算法的特征：
- 1.求解问题是多阶段决策问题；
- 2.求解过程是多步判断（多个子问题），每步求解的问题是后面阶段求解问题的子问题，最后一步求解出原问题的解；
- 3.前面一个子问题的最小值与后面问题的最小值有依赖关系。

$$F(B_2) = \min\{B_2C_1 + F(C_1), B_2C_2 + F(C_2)\}$$

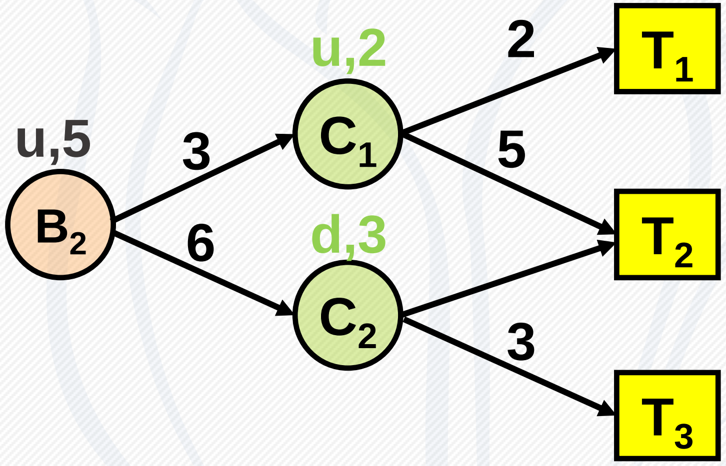
PART 02

动态规划算法的必要条件

Dynamic Programming

动态规划的必要条件

优化函数的特点： 任何最短路径的子路径都是相对于子路径始点和终点的最短路径



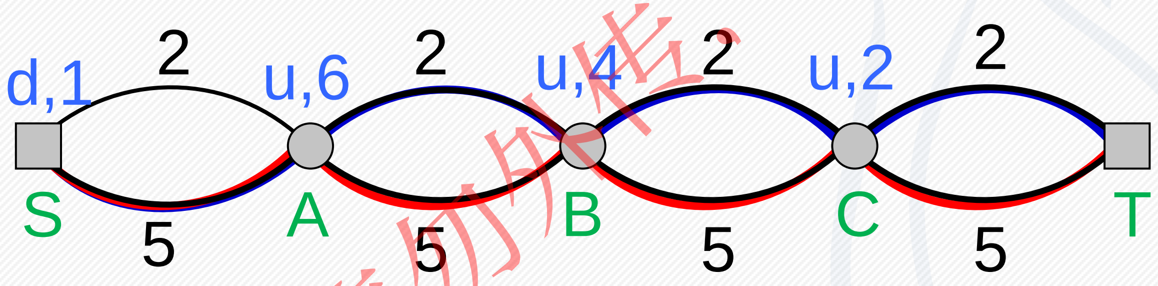
优化原则（最优子结构性质）

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

分析和证明问题的最优子结构性质时，一般采用反证法

动态规划的必要条件

例：求总长模10的最小路径



动态规划算法的解：下、上、上、上； ≠ 实际最优解：下、下、下、下；

实际最优解不满足优化原则：

全局最优路径在A->T和C->T的子问题中不是最优；

因此，不能使用动态规划算法。

动态规划的设计要素

- 1. 如何划分子问题（边界）？
- 2. 原问题的优化函数值与子问题的优化函数值存在着什么依赖关系？
- 3. 是否满足优化原则？
- 4. 递推方程是什么？初值等于什么？

PART 03

动态规划算法的递归实现

Dynamic Programming

矩阵A： i行j列，矩阵B: j行k列

以元素相乘作基本运算，计算 $C=AB$的工作量

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \\ b_{21} & b_{22} & \cdots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{j1} & b_{j2} & \cdots & b_{jk} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1k} \\ c_{21} & c_{22} & \cdots & \boxed{c_{2k}} \\ \vdots & \vdots & \ddots & \vdots \\ c_{i1} & c_{i2} & \cdots & c_{ik} \end{bmatrix}$$

$$c_{mn} = \sum_{l=1}^{j} a_{ml} b_{ln}$$

矩阵C: i行 k列，元素个数为 ik，计算每个元素需要做 j次乘法和j-1次加法;

一次矩阵乘法中，元素乘法次数为 $i\,j\,k$, 乘法和加法次数为 $i\,j\,(2k-1)$

一个实例

- 实例： P = <10, 100, 5, 50>
- A_1: 10 × 100, A_2: 100 × 5, A_3: 5 × 50

$$\overbrace{(\underbrace{(A_1 \times A_2)}_{10 \times 100 \times 5 = 5000} \times A_3)}^{10 \times 5 \times 50 = 2500}$$

$$\overbrace{(A_1 \times (\underbrace{A_2 \times A_3}_{100 \times 5 \times 50 = 25000}))}^{10 \times 100 \times 50 = 50000}$$

- 乘法次序：
- $((A_1 A_2)A_3)$: 10 × 100 × 5 + 10 ×5 × 50 = 7500 ✓
- $(A_1(A_2 A_3))$: 10 × 100 × 50 + 100 × 5 × 50 = 75000

为什么3个矩阵要用2个括号？

例3.3 矩阵链相乘

- 问题：设$A_1, A_2, \ldots, A_n$为矩阵序列，A_i为 $P_{i-1} \times P_i$阶矩阵，$i = 1, 2, \ldots, n$.　确定乘法次序使得元素相乘的总次数最少。
- 输入：向量 $P = \langle P_0, P_1, \ldots, P_n \rangle$，其中，$P_0, P_1, \ldots, P_n$是$n$个对应矩阵的行数和列数。
- 输出：使得元素相乘的总次数最少的矩阵链乘法加括号的位置。

列举出所有可能的计算次序，从中找出数乘次数最少的情况

算法复杂度分析

每加一个括号原问题就可以分解为两个子矩阵的加括号问题。

假设n个矩阵的连乘问题中，不同的计算次序总数为P(n)。假设最后一次相乘发生在第k(1≤k≤n-1)个矩阵的位置，即$(A_1 \ldots A_k)(A_{k+1} \ldots A_n)$，则P(n)= P(k)P(n-k). 因此，P(n)可表示为:

$$P(n) = \begin{cases} 0, & n=1; \\ \sum_{k=1}^{n-1} P(k)P(n-k), & n>1. \end{cases}$$

n个矩阵相乘，需要加n-1对括号，P(n) 是一个Catalan数 C(n-1):

$$P(n) = \frac{1}{n} C_{n-1}^{2(n-1)} = \Omega(4^n / n^{3/2})$$

子问题划分

- 问题：$A_{i\,..\,j}$ $(i<j)$表示矩阵链 $A_iA_{i+1}\ldots A_j$ 的乘积，矩阵的维数存储向量： $P=<P_{i-1}, P_i, \ldots, P_j>$

- 假设已知最优划分的运算次数为$m[i, j]$，并且最后一次相乘发生在第k个矩阵的位置，即

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

- 只有在矩阵子链 $A_{i\,..\,k}$和$A_{k+1\,..\,j}$取得最优次序（子问题最优解）时，$A_{i\,..\,j}$才能取得最优次序（原问题的最优解）

优化原则证明

- 已知原问题$A_{i..j}$最优划分导致的运算次数为$m[i, j]$，那么原问题$A_{i..j}$的最优次序（原问题的最优解）所包含的子问题$A_{i..k}$和$A_{k+1..j}$中的次序也是最优的（子问题最优解）。 $f[i,k]$ $f[k+1, j]$

反证法：

假设子问题$A_{i..k}$存在（比原问题的最优解在该问题上的解）更优的解，其运算次数为$f'[i, k](<f[i, k])$，则原问题一个新的可行解（$A_{i..k}$"新解"+ $A_{k+1...j}$ "老解"），其运算次数为：

$$f'[i, j] = f'[i,k] + f[k+1, j] + P_{i-1}P_kP_j$$

$f'[i, j] < m[i, j]$，与已知矛盾。

含弘光大　继往开来

递推方程

- 已知原问题$A_{i..j}$最优划分导致的运算次数为$m[i,j]$，那么原问题$A_{i..j}$的最优次序（原问题的最优解）所包含的子问题$A_{i..k}$和$A_{k+1..j}$中的次序也是最优的（子问题最优解），因此

$$m[i,j]=m[i,k]+m[k+1,j]+P_{i-1}P_kP_j$$

- **递推方程和初值（最后一次相乘的位置k未知）：**

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

遍历所有可能划分的位置

递归实现

设立标记函数

为了追踪最优解，设计表 $s[i,j]$，记录求得子问题最优运算次序中最后一次乘法的位置

算法3.1 RecurMatrixChain (P, i, j) 首次调用，i←1, j ←n

1. *if i==j then*
2.   *m[i, j]←0; s[i, j]←i; return m[i, j]*
3. *end*
4. *m[i, j]←∞; s[i, j]←i;*
5. for *k←i* to *j*–1 do
6. $q \leftarrow$ RecurMatrixChain(P, i, k)+ RecurMatrixChain(P, k+1, j) + $p_{i-1}p_kp_j$;
7.   *if q < m[i, j] then*
8.   *m[i, j]←q; s[i, j]←k;*
9.   *end*
10. *end*
11. return *m[i, j]*

单个矩阵，直接返回

划分位置$k=i$

遍历所有可能划分的位置

找到更好的解

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

含弘光大 继往开来

时间复杂度的递推方程

$$T(n) \ge \begin{cases} 0 & n = 1 \\ \sum_{k=1}^{n-1}(T(k) + T(n-k) + O(1)) & n > 1 \end{cases}$$

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

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

定理3.1：对于$n>1$，$T(n)=\Omega(2^{n-1})$

数学归纳法证明.

当$n=2$，$T(2)\geq c = c_1 2^{2-1}, c_1 = c/2$为某个正数

假设对于任何小于n 的 k 命题为真，则存在c'使得

$$T(n)\geq c'n+2\sum_{k=1}^{n-1}T(k)$$ 代入归纳假设

$$\geq c'n+2\sum_{k=2}^{n-1}c_1 2^{k-1}$$ 等比数列求和

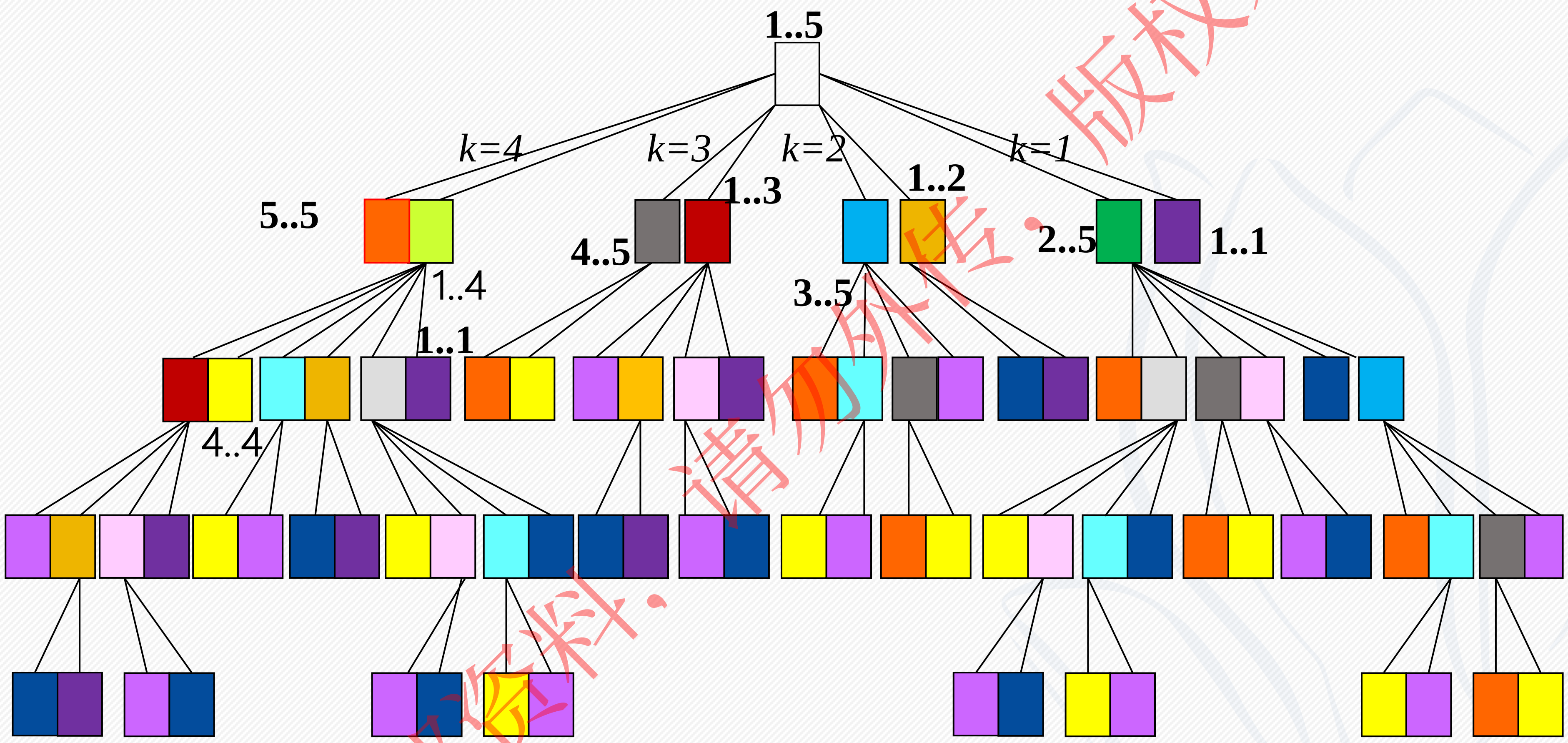
$$=c'n+c_1(2^n-4)\geq c_1 2^{n-1}$$

PART 04

动态规划算法的递归+备忘录实现

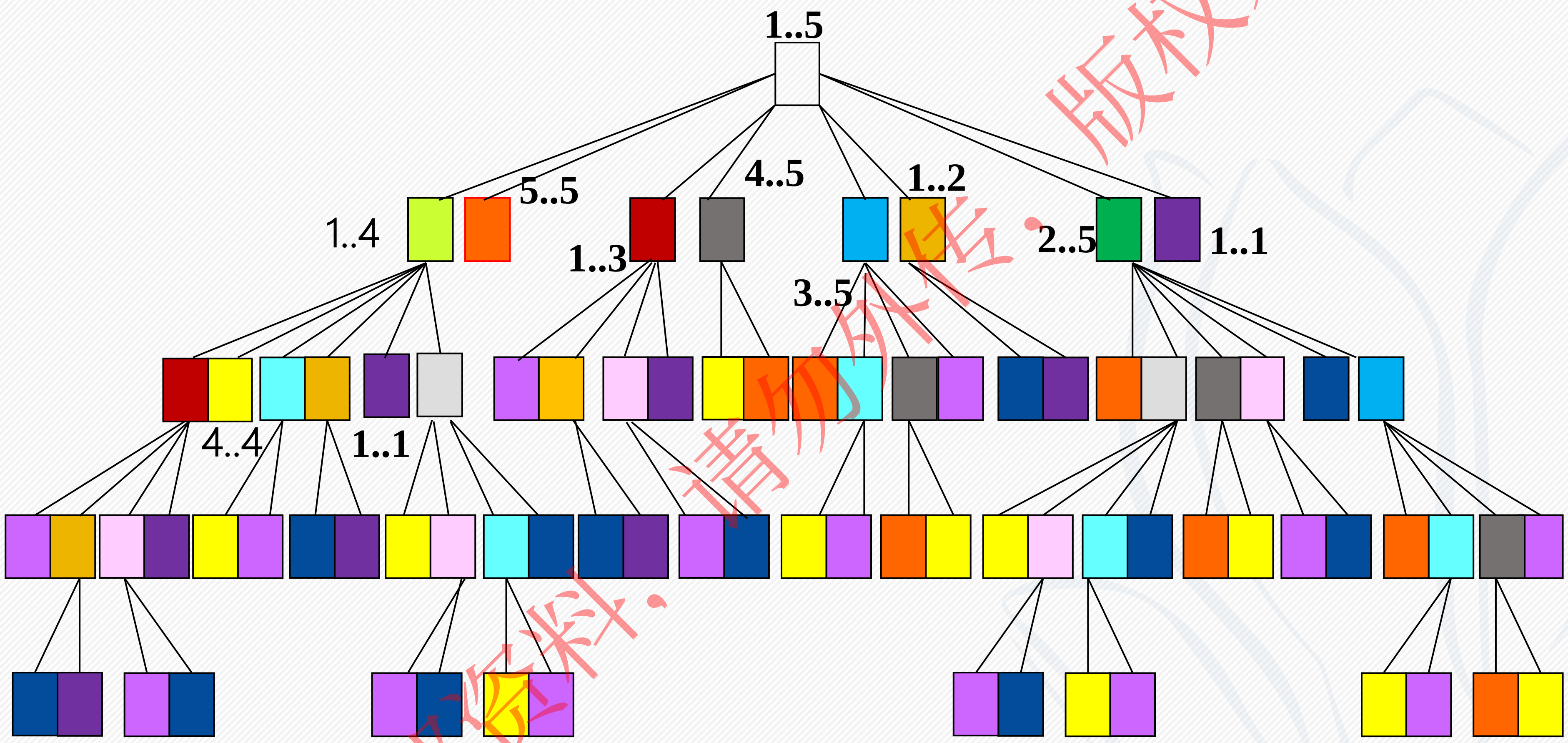
Dynamic Programming

子问题的产生



***n*=5，第一层8个子问题，节点8个；**
子节点共计：81个；

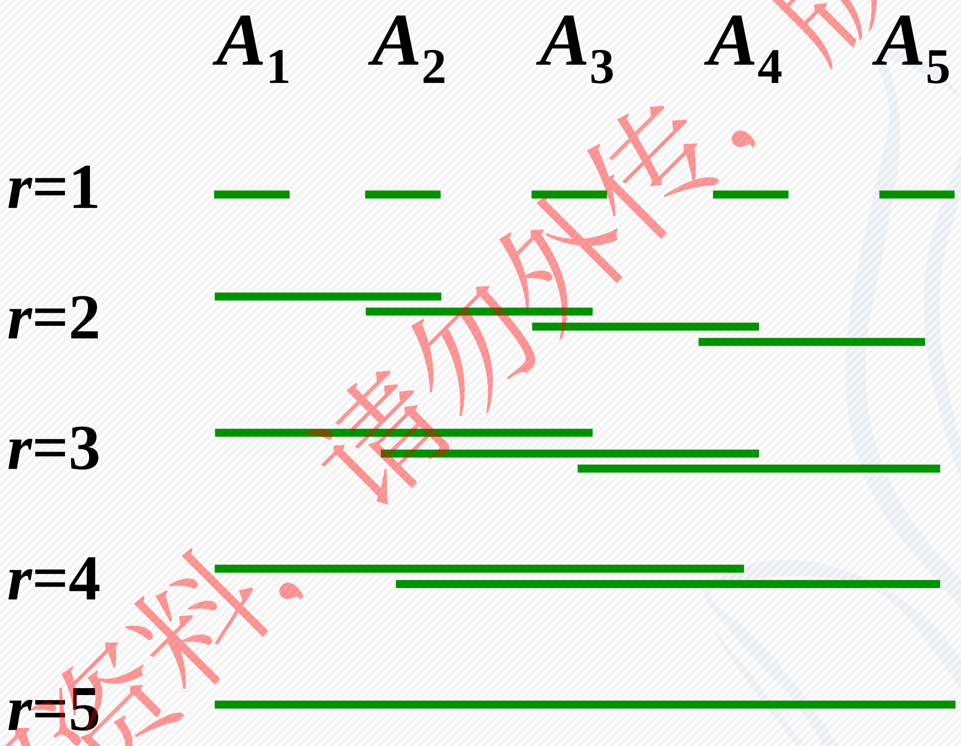
子问题的产生



***n*=5，第一层8个子问题，节点8个；**
子节点共计：81个；

子问题的类别

$n=5$



子问题	1-1	2-2	3-3	4-4	5-5
个数	**8**	**12**	**14**	**12**	**8**
子问题	1-2	2-3	3-4	4-5	
个数	**4**	**5**	**5**	**4**	
子问题	1-3	2-4	3-5		
个数	**2**	**2**	**2**		
子问题	1-4	2-5			
个数	**1**	**1**			
子问题	1-5				
个数	**1**				

***n*=5，计算子问题：81个；**
不同的子问题：15个

动态规划算法的递归实现效率不高，
原因：同一子问题多次重复出现，
每次出现都需要重新计算一遍

Those who cannot remember the past are doomed to repeat it.

-----George Santayana,
The life of Reason: Introduction and Reason in Common Sense

- 如果保存已解决的子问题的答案，在需要时直接查找和调用，就可以避免重复计算，提高算法效率。

- 改进途径一：
- 开辟一个存储空间（“**备忘录**”），记录子问题的**划分边界（标记函数，最优解）**和**优化函数值（元素计算次数，最优值）**。
- 计算子问题时，先检索备忘录，如有就直接调用，否则计算并记录。
- 通过备忘录提高效率的同时，增大了空间开销。

首次调用，i←1, j ←n, m←0, s[i][j]←0

设立标记函数

为了追踪最优解，设计表 $s[i,j]$，记录求得子问题最优运算次序中最后一次乘法的位置

```
lookupChain(p, i, j)
if (i == j) then
    return 0
end
if m[i][j] > 0 then
    return m[i][j]
end
u ←0+ lookupChain(p,i+1,j) + p[i-1]*p[i]*p[j];
s[i][j] ← i;
for k ← i+1 to j-1 do
    t ← lookupChain(p,i,k) + lookupChain(p,k+1,j) + p[i-1]*p[k]*p[j];
    if (t < u) then
        u ← t; s[i][j] ← k;
    end
end
m[i][j] ← u;      // m矩阵保存在内存中
return m[i][j]
```

单个矩阵，直接返回

子问题若已经计算过，则直接查找后返回

如果最后划分位置k取i

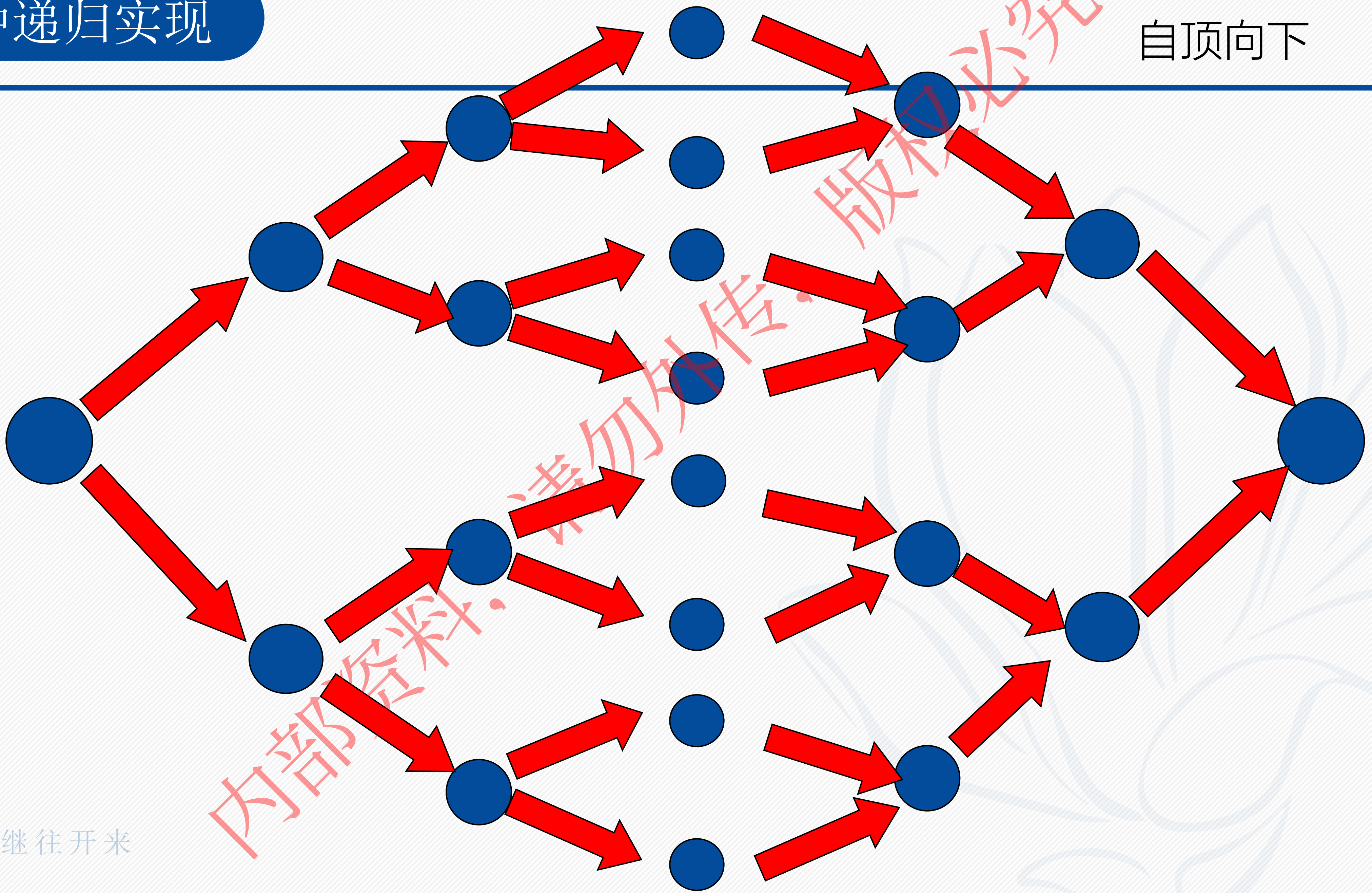
找到更好的划分，就替换原有的值

遍历所有可能划分的位置

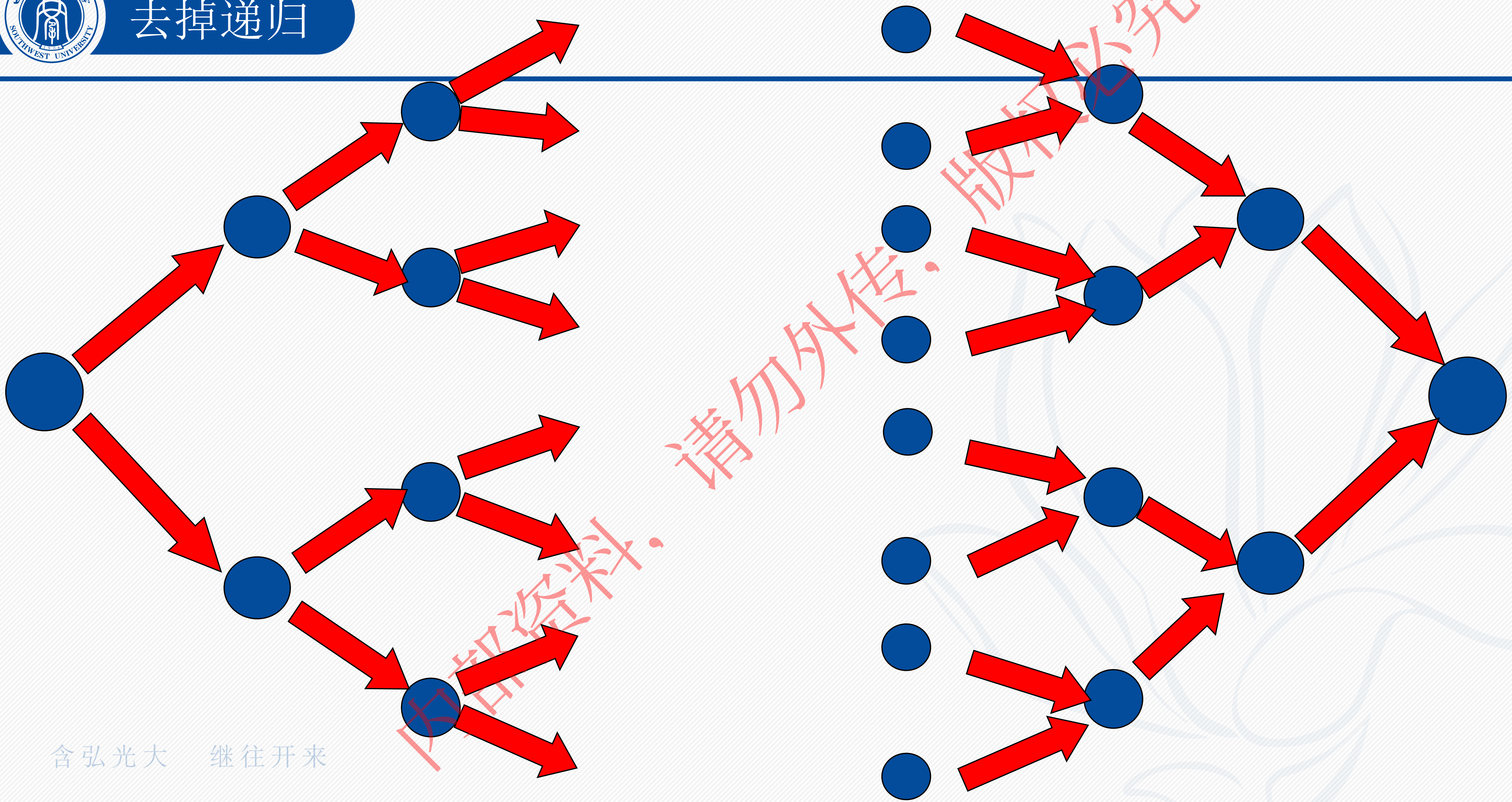
$$m[i,j]=\begin{cases}0; & i=j\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\}; & i<j\end{cases}$$

两种递归实现

自顶向下



去掉递归



PART 05

动态规划算法的迭代实现

Dynamic Programming

迭代实现

- 改进途径二：
- 设计自底向上的求解过程（计算顺序），保证后面用到的值前面已经计算好。
- 从最小规模（只有一个矩阵，最小子问题）的问题开始算起，直到规模为n的子问题（原问题），保证每个问题只计算一次。
- 将已求解的小规模问题的解合并成为一个更大规模问题的解，自底向上逐步得到原问题解

迭代实现

- 子问题的类别
- 长度为1： 只含1 个矩阵，有n 个不同的子问题(不需要计算)；
- 长度为2： 含2 个矩阵，n-1 个不同的子问题；
- 长度为3： 含3 个矩阵，n-2 个不同的子问题；
- …
- 长度为n-1： 含n-1 个矩阵，2 个不同的子问题；
- 长度为n： 原始问题，只有1 个；

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；
- 每个问题中最后一次相乘发生的位置有1种情况
- 长度为3 ： 1..3, 2..4, 3..5, ..., n-2..n；
- 每个问题中最后一次相乘发生的位置有2种情况
- ...
- 长度为n-1： 1.. n-1, 2..n；
- 长度为n ： 1.. n；

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

n=5时的实例

$$\min_{i\le k<j}\{m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\}$$

子问题	1-1	2-2	3-3	4-4	5-5
子问题	1-2	2-3	3-4	4-5	
子问题	1-3	2-4	3-5		
子问题	1-4	2-5			
子问题	1-5				

k=3, 子问题: 3..3, 4..5

k=4, 子问题: 3..4, 5..5

k=2, 子问题: 2..2, 3..5

k=3, 子问题: 2..3, 4..5

k=4, 子问题: 2..4, 5..5

迭代实现

时间复杂度：

行2,3,7循环都是 $O(n)$，循环内为 $O(1)$，

因此 $W(n) = O(n^3)$

算法3.2 MatrixChain(P, n)

```
1.  令所有的 m[i, i]初值为0      1≤ i ≤ n
2.  for  r ←2  to  n  do                         // r为矩阵链长度(子问题规模)
3.      for i ←1  to  n–r+1  do                  //子问题的左边界
4.          j ← i +r–1;                          //子问题的右边界
5.          m[i, j] ← 0+m[i+1, j] + p_{i–1}*p_i*p_j ;  //分割位置为i时的计算量
6.          s[i, j] ← i;                         //记录分割位置
7.          for  k ← i +1  to  j–1  do           //分割位置k向右移动
8.              t ← m[i, k]+m[k+1, j]+ p_{i–1}*p_k*p_j ;
9.              if  t < m[i, j] then
10.                 m[i, j]← t; s[i, j]← k;      //更新结果
11.             end
12.         end
13.     end
14. end
```

遍历所有规模为r的子问题

遍历所有可能的分割

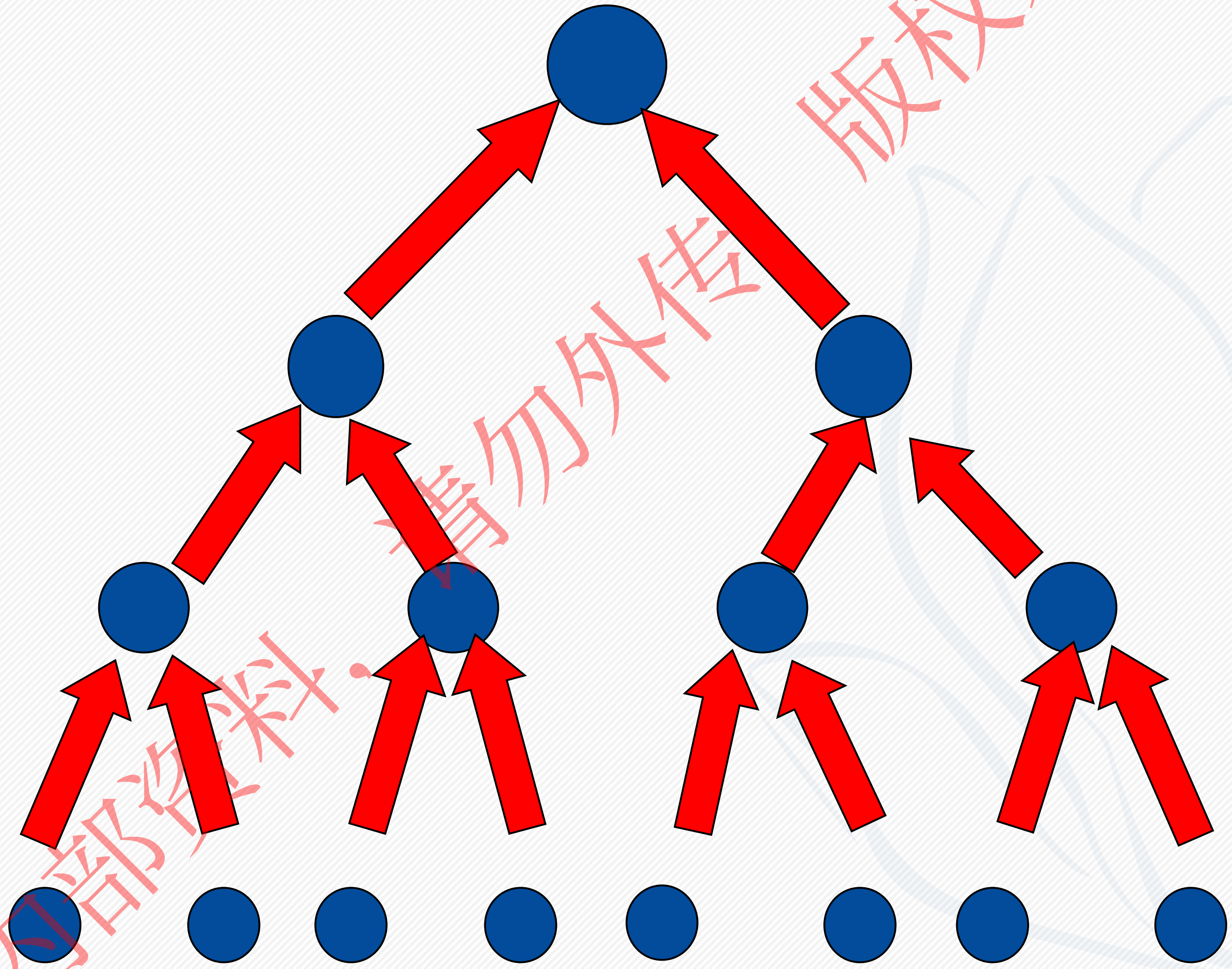
设立标记函数

为了追踪最优解，设计表 $s[i,j]$，记录子问题**最优运算次序**中**最后一次乘法**的位置

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

迭代实现

自底向上



一个实例

- 输入： $P = <30, 35, 15, 5, 10, 20>, n=5$；

- 矩阵链：$A_1\, A_2\, A_3\, A_4\, A_5$，其中
- A_1:30×35，A_2:35×15，A_3:15×5，A_4:5×10，A_5:10×20

- 备忘录：存储所有子问题的最小乘法次数（最优目标函数
- 值）
- 标记函数：记录最优解或者最优解的部分分量，即得到最优目标函数值的划分位置.

一个实例

P = <30, 35, 15, 5, 10, 20>

$$\min_{i \le k < j}\{m[i,k]+m[k+1,j]+P_{i-1}P_kP_j\}$$

备忘录m[i,j]

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				

k=2, 子问题: 2..2, 3..5 (35, 15) (15, 20)

k=3, 子问题: 2..3, 4..5 (35, 5) (5, 20)

k=4, 子问题: 2..4, 5..5 (35, 10) (10, 20)

m[2,5] = min{ 0+2500+35×15×20, 2625+1000+35×5×20, 4375+0+35×10×20} = 7125

s[2,5]=3

s[i,j]，记录求得最优运算次序中最后一次乘法的位置

一个实例

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：$s[1,5]=3 \Rightarrow (A_1A_2A_3)(A_4A_5)$

$s[1,3]=1 \Rightarrow A_1(A_2A_3)$

输出：$((A_1(A_2A_3))(A_4A_5))$ ✓

计算顺序：$(A_1(A_2A_3))(A_4A_5)$ ✗

最少的乘法次数：$m[1,5]=11875$

三种实现的比较

- 动态规划算法的递归实现
  - ➢采用自顶向下的策略，从最大规模问题开始；
  - ➢子问题被多次重复计算，子问题计算次数呈指数增长；
- 动态规划算法的递归+备忘录实现
  - ➢采用自顶向下的策略，从最大规模问题开始；
  - ➢子问题只被计算一次，但是被多次重复调用，调用次数呈指数增长；
- 动态规划算法的迭代实现
  - ➢采用自底向上的策略，从最小规模问题开始；
  - ➢每个子问题只计算一次，子问题的计算随问题规模成多项式增长。
- 动态规划算法，默认使用迭代实现的方式

动态规划算法的设计步骤

01

划分子问题：

用参数表达子问题的边界，将问题求解转变成多步判断的过程

02

确定优化函数：

以该函数的极大(或极小)作为判断的依据，确定原问题和子问题是否满足优化原则

03

列出递推方程：

列出关于优化函数的递推方程(或不等式)和边界条件

04

设立标记函数：

考虑是否需要设立标记函数

05

存储备忘录：

自底向上计算，以备忘录方法(表格)存储中间结果

06

追踪最优解：

根据备忘录和优化函数追踪原问题的最优解

总结

- 1.动态规划的思路：把求解多阶段决策问题转化为一系列单阶段决策问题（子问题与单阶段）；
- 2.动态规划的适用条件：最优子结构性质；
- 3.动态规划的递归实现效率不高，原因是同一子问题多次重复计算；
- 4.动态规划的设计要素：界定子问题的边界，原问题与子问题优化函数的关系，优化原则的证明，递推方程及初值，标记函数和备忘录。

- 动态规划算法与分治法
  - 将待求解问题**拆分**成若干个子问题，然后根据子问题的解**归并**得到原问题的解。

- 不同点：
  - 1. 分治算法**自顶向下**求解，动态规划算法**自底向上**求解；
  - 2.分治算法的前提是子问题与原始问题的性质一样、最小子问题可以求解、子问题相互独立（求解）；动态规划算法的前提是满足子问题与原始问题的性质一样、最小子问题可以求解、最优子结构；
  - 3.分治法将分解后的子问题看成**相互独立**的，通过用**递归**来实现拆分和归并；动态规划分解后的子问题有**重叠部分**，需要**存储备忘录**避免重复计算，默认用**迭代**来实现。

PART 06

投资问题

Dynamic Programming

投资问题

实例

➢ 5万元钱，4个项目，效益函数如下表所示：

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

含弘光大　继往开来

■ 已知：m元钱，n项投资，每个项目上投资额度是非负整数

■ 求：使得总效益最大的投资方案

问题描述：

■ 设x_i是投给项目i的钱数(i∈(1,n))

■ $f_i(x_i)$: 将x_i元投入第i个项目的效益

优化问题：

$$\max \{f_1(x_1) + f_2(x_2) + \dots + f_n(x_n)\}$$

$$s.t.: x_1 + x_2 + \dots + x_n = m, \quad x_i \in N$$

投资问题

划分子问题

仿照最短路径和矩阵链乘法，可投资的项目和投资总钱数会影响总收益，据此划分子问题

子问题界定：

- 参数k：允许对前k个（1, 2, ..., k ）项目投资
- 参数x：投资总钱数为 x

子问题计算顺序：

- 第一层： k = 1, 2, ... , n;
- 第二层：对于给定的 k，x = 1, 2, ... , m;

含弘光大　继往开来

确定优化函数

- 假设$F_k(x)$表示x元钱投给前k个项目的可以达到的最大效益，对应的最佳投资方案中第k个项目的投资金额为x_k元，则：

$$F_k(x) = f_k(x_k) + F_{k-1}(x - x_k)$$

- 只有在子问题（前k-1个项目，总投资金额为$x - x_k$元）取得最优解时，原问题（前k个项目，总投资金额为x元）才能取得最优解。

投资问题

划分子问题

仿照最短路径和矩阵链乘法，可投资的项目和投资总钱数会影响总收益，据此划分子问题

子问题界定：

- 参数k：允许对前k个（1, 2, ..., k）项目投资
- 参数x：投资总钱数为 x

子问题计算顺序：

- 第一层：k = 1, 2, ... , n;
- 第二层：对于给定的 k，x = 1, 2, ... , m;

含弘光大　继往开来

确定优化函数

- 假设$F_k(x)$表示x元钱投给前k个项目的可以达到的最大效益，对应的最佳投资方案中第k个项目的投资金额为x_k元，则：

$$F_k(x) = f_k(x_k) + F_{k-1}(x - x_k)$$

反证法：

假设子问题（前k-1个项目，总金额为$(x-x_k)$）存在（比全局最优解的部分分量）更优的解，达到的效益为$F'_{k-1}(x-x_k)(>F_{k-1}(x-x_k))$，由此（“老解”+“新解”）构造出原问题一个新的可行解，其效益为：

$$F'_k(x) = f_k(x_k) + F'_{k-1}(x - x_k)$$

$F'_k(x) > F_k(x)$，与已知矛盾。

投资问题

划分子问题

仿照最短路径和矩阵链乘法，可投资的项目和投资总钱数会影响总收益，据此划分子问题

子问题界定：

- 参数k：允许对前k个（ 1, 2, ..., k ）项目投资
- 参数x：投资总钱数为 x

子问题计算顺序：

- 第一层： k = 1, 2, ... , n;
- 第二层：对于给定的 k，x = 1, 2, ... , m;

确定优化函数

- 设$F_k(x)$表示x元钱投给在前k个项目的可以达到的最大效益，对应的最佳投资方案中，投给第k个项 目x_k元，则：

$$F_k(x) = f_k(x_k) + F_{k-1}(x - x_k)$$

列出递推方程

若x_k未知，遍历其所有可能取值：

$$F_k(x) = \max_{0 \le x_k \le x}\{ f_k(x_k) + F_{k-1}(x - x_k)\} \quad k > 1$$

$$F_1(x) = f_1(x)$$

投资问题

$x_k(x)$为标记函数

$x_k(x)=h$表示在子问题（前k个项目投资，总投资金额x元钱）取得最优解（取得了最大收益$F_k(x)$）时，在第 k 个项目上的投资金额为h元.

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

$k=1$为初值

$x_1(x)=x,\quad x=1,2,3,4,5;$

$F_1(1)=11, F_1(2)=12, F_1(3)=13,$

$F_1(4)=14, F_1(5)=15,$

$k=2$　　$F_2(x)=\max\limits_{0\le x_2\le x}\{f_2(x_2)+F_1(x-x_2)\}$

x	$f_1(x)$	$f_2(x)$
0	0	0
1	11	0
2	12	5
3	13	10
4	14	15
5	15	20

x	$F_1(x)$	$x_1(x)$
1	11	1
2	12	2
3	13	3
4	14	4
5	15	5

$x_2(x)$=m表示在子问题（前2个项目投资，总投资金额x元钱）取得最优解时，在第 2个项目上的投资金额为m元.

$F_2(1)=\max\{f_2(0)+F_1(1),\ f_2(1)+F_1(0)\}=11$；　$x_2(1)=0$;

$F_2(2)=\max\{f_2(0)+F_1(2),\ f_2(1)+F_1(1),\ f_2(2)+F_1(0)\}=12$;

$x_2(2)=0$;

$F_2(3)=\max\{f_2(0)+F_1(3),\ f_2(1)+F_1(2),$

$f_2(2)+F_1(1),\ f_2(3)+F_1(0)\}=16$;

$x_2(3)=2$;

k = 3 $F_3(x)=\max\limits_{0\le x_3\le x}\{f_3(x_3)+F_2(x-x_3)\}$

x	$F_1(x)$	$x_1(x)$	$F_2(x)$	$x_2(x)$
1	11	1	11	0
2	12	2	12	0
3	13	3	16	2
4	14	4	21	3
5	15	5	26	4

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

$F_3(1)=\max\{f_3(0)+F_2(1),\ f_3(1)+F_2(0)\}\qquad x_3(1)=0;$

$F_3(2)=\max\{f_3(0)+F_2(2),\ f_3(1)+F_2(1),\ f_3(2)+F_2(0)\}=12;\quad x_3(2)=1;$

$F_3(3)=\max\{f_3(0)+F_2(3),\ f_3(1)+F_2(2),$

$f_3(2)+F_2(1),\ f_3(3)+F_2(0)\}=16;\quad x_3(3)=3;$

实例的计算

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

$$x_4(5)=1 \Rightarrow x_4=1, x_3(5-1)=x_3(4)$$

$$x_3(4)=3 \Rightarrow x_3=3, x_2(4-3)=x_2(1)$$

$$x_2(1)=0 \Rightarrow x_2=0, x_1(1-0)=x_1(1)$$

$$x_1(1)=1 \Rightarrow x_1=1$$

解 $x_1=1$, $x_2=0$, $x_3=3$, $x_4=1$ $F_4(5)=61$

算法的复杂度分析

备忘录中有 m 行 n 列, 共计 mn 项$F_k(x)$ 和$x_k(x)$ （与$F_k(x)$ 同时计算出）。

$$F_k(x)=\max_{0\le x_k\le x}\{f_k(x_k)+F_{k-1}(x-x_k)\} \quad k>1$$

$$F_1(x)=f_1(x)$$

x_k 有$(x+1)$种可能性，计算每一个项$F_k(x)$ $(2\le k\le n, 1\le x\le m)$ 时，要 $x+1$ 次加法, x 次比较。

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

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

$$W(n)=O(nm^2)$$

PART 07

背包问题

Dynamic Programming

例3.5 背包问题

- 一个旅行者随身携带一个背包. 可以放入背包的物品有n种, 每种物品的重量和价值分别为 w_j, v_j. 如果背包的最大重量限制是 b，并且每种物品可以放多个。
- 怎样选择放入背包的物品以使得背包的价值最大?

假设x_j 是装入背包的第 j种物品个数

目标函数 $$\max \sum_{j=1}^{n} v_j x_j$$

约束条件 $$\sum_{j=1}^{n} w_j x_j \le b,$$

$$x_j \in \mathbf{N}$$

线性规划问题：约束条件和目标函数都是线性函数。

整数规划问题：线性规划问题的变量 x_j 都是非负整数。

0-1背包问题：x_j=0或1。

子问题界定和计算顺序

- 子问题界定：允许选择的物品种类和允许装入的重量
- k：考虑对物品1, 2, … , k 的选择
- y：背包总重量不超过 y

- 子问题计算顺序：
- 第一步：k = 1, 2, … , n
- 第二步：对于给定的 k，y = 1, 2, … , b

- 原问题：k = n, y = b

优化原则

- $F_k(y)$：可以装前 k 种物品，总重不超过 y，背包的**最大价值**
- 假设已知$F_k(y)$和最佳方案中的$\mathbf{x_k}$，原问题$F_k(y)$和子问题$F_{k-1}(*)$满足什么关系？

$$F_k(y) = x_k v_k + F_{k-1}(y - w_k x_k)$$

反证法：

假设子问题（前k-1类物品可选，重量限制为$(y-w_k x_k)$）存在（比全局最优解的部分分量）更优的解，达到的效益为$F'_{k-1}(y-w_k x_k)(>F_{k-1}(y-w_k x_k))$，由此（"老解"+"新解"）构造出原问题一个新的可行解，其价值为：

$$F'_k(y) = x_k v_k + F'_{k-1}(y - w_k x_k)$$

$F'_k(x) > F_k(x)$，与已知矛盾。

优化原则

- $F_k(y)$：可以装前 k 种物品，总重不超过 y，背包的**最大价值**

- 假设已知$F_k(y)$和最佳方案中的$\mathbf{x_k}$，原问题$F_k(y)$和子问题$F_{k\text{-}1}(*)$满足什么关系？

$$F_k(y) = x_k v_k + F_{k-1}(y - w_k x_k)$$

- 递推方程如何表示？

$$F_k(y) = \max_{0 \le x_k \le \lfloor y/w_k \rfloor} \{x_k v_k + F_{k-1}(y - w_k x_k)\}$$

递推方程

- 子问题只考虑两种情况：
- 第k种物品装($\boldsymbol{x_k}$≥1)或者不装($\boldsymbol{x_k}$ =0)

$$F_k(y)=\max\{\boxed{F_{k-1}(y)},\boxed{F_k(y-w_k)}+v_k\}$$

$$F_0(y)=0,\quad 0\le y\le b;$$

先放1个k物品

$$F_k(0)=0,\quad 0\le k\le n$$

$$F_1(y)=\left\lfloor\frac{y}{w_1}\right\rfloor v_1$$

$$F_k(y)=-\infty\quad y<0$$

剩余空间不足以装1个k物品

- 夺命三连问：
- 1.对么？
- 2.可行（可计算）么？
- 3.为什么这样构造？

标记函数

翻译翻译：$i_5(8)=3$

- $i_k(y)=m$的含义
- 允许选择前 k 种物品，总重不超过 y，背包达最大价值时装入物品的**最大标号为m**；

装入第k种物品效果不好$x_k=0$

$$i_k(y)=\begin{cases} i_{k-1}(y) & F_{k-1}(y) > F_k(y-w_k)+v_k \\ k & F_{k-1}(y) \le F_k(y-w_k)+v_k \end{cases}$$

装入第k种物品效果好$x_k\ge 1$

$$i_1(y)=\begin{cases} 0 & y < w_1 \\ 1 & y \ge w_1 \end{cases}$$

实例计算

$v_1 = 1,\ \ v_2 = 3,\ \ v_3 = 5,\ \ v_4 = 9,\ w_1 = 2,\ \ w_2 = 3,\ \ w_3 = 4,\ \ w_4 = 7,\ b = 10$

$F_k(y)$ 的计算表如下:

k \ y	1	2	3	4	5	6	7	8	9	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

用标记函数追踪问题的解

$v_1=1,\ v_2=3,\ v_3=5,\ v_4=9,\ w_1=2,\ w_2=3,\ w_3=4,\ w_4=7,\ b=10$

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

$i_4(10)=4 \Rightarrow x_4 \ge 1$

设$x_4=1$，$i_4(10-w_4)=i_4(3)=2<4 \Rightarrow x_4=1$，$x_3=0,\ x_2 \ge 1$

设$x_2=1$，$i_2(3-w_2)=i_2(0)=0<2 \Rightarrow x_2=1$，$x_1=0$

解 $x_1=0,\ x_2=1,\ x_3=0,\ x_4=1$

追踪解 x_j 的算法

算法3.3 TrackSolution

输入：$i_k(y)$表，其中k=1,2,…,n，y=1,2,…,b

输出：$x_1, x_2, \ldots, x_n$，n种物品的装入量

1. for $j \leftarrow 1$ to n do
2. $\quad x_j \leftarrow 0$;
3. end
4. $y \leftarrow b; j \leftarrow n$; //原问题
5. $j \leftarrow i_j(y)$; //初始追踪
6. $x_j \leftarrow 1; y \leftarrow y - w_j$； //装1个物品j
7. while $i_j(y) == j$ do； //是否继续装物品j
8. $\quad y \leftarrow y - w_j;\ x_j \leftarrow x_j + 1$;
9. end
10. if $i_j(y) \neq 0$ then //问题是否结束
11. $\quad$ goto 5; //继续装其他物品
12. end

根据公式

$$F_k(y)=\max\{F_{k-1}(y),F_k(y-w_k)+v_k\}$$

备忘录需计算nb项$F_k(y)$，每项常数时间（一次加法和一次比较）

计算时间为$O(nb)$，伪多项式时间.

当背包容量b很大时，算法需要的计算时间较多。
例如，当$b>2^n$时，算法需要$\Omega(n2^n)$计算时间。