算法设计与分析

讲授内容：**动态规划**

2025年11月21日

第15章 动态规划

■ 目录

- ➤ 动态规划基本思想
- ➤ 动态规划求解步骤
- ➤ 动态规划问题举例
  - 钢条切割问题
  - 矩阵链乘法问题
  - 动态规划原理
  - 最长公共子序列问题
  - 装配线调度问题
  - 最优二叉搜索树问题

引例：

- 针对斐波那契数列

$F(0)=0$

$F(1)=1$

$F(n)=F(n-1)+F(n-2)$

问题：

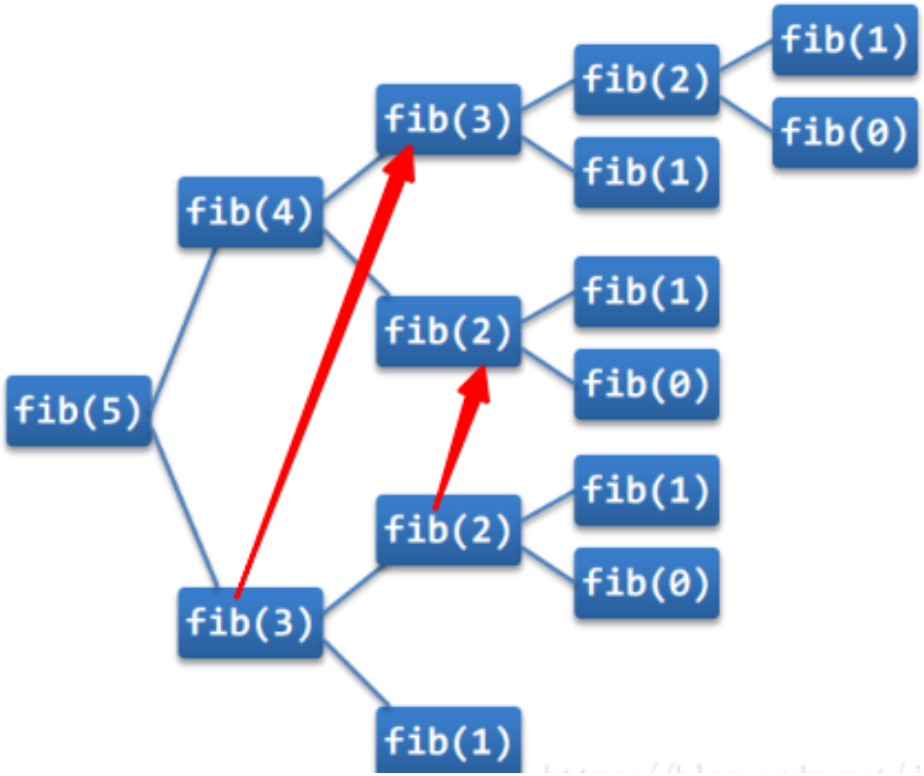
（1）请计算F(10)， F(12)，和F(15)

（2）请给出计算F(*n*)的算法思想和伪代码。

引例：

- 递归法求解

```
function Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)
```



- 优点：实现简单
- 缺点：存在大量重复计算，导致时间复杂度较高（约为 $O(2^n)$）

引例

- 增加备忘录

```
function Fibonacci(n, memo):
    if n in memo:
        return memo[n]
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        memo[n] = Fibonacci(n - 1, memo) + Fibonacci(n - 2, memo)
        return memo[n]
```

- 通过使用备忘录，避免了重复计算，显著提高了效率

引例

- 自底向上地计算

```
function Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    # 创建一个数组 dp 来存储计算结果
    dp = array of size (n + 1)

    # 初始化前两个斐波那契数
    dp[0] = 0
    dp[1] = 1

    # 从 2 开始计算到 n
    for i from 2 to n:
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

动态规划(Dynamic programming)

- Programming含义：指的是一种表格法，而非计算机程序
- 基本思想：和分治法类似，都是通过组合子问题的解而得到整个问题的解
  - 所不同的是，动态规划法适用于子问题重叠的情况
  - 也就是不同的子问题具有公共的子问题
  - 子问题的求解递归进行，将其划分为更小的子子问题
- 通常用来求解最优化问题

动态规划的基本思想

- 动态规划的实质是分治思想和解决冗余
  - **分治**：将问题实例分解为更小的、相似的子问题
  - **解决冗余**：存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。

- 所解决问题的特征：子问题呈现大量的重复。

- 动态规划法的关键：**对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来；以后再遇到时直接引用，不必重新求解。**

习题1：钢条切割问题

◆ 问题描述：

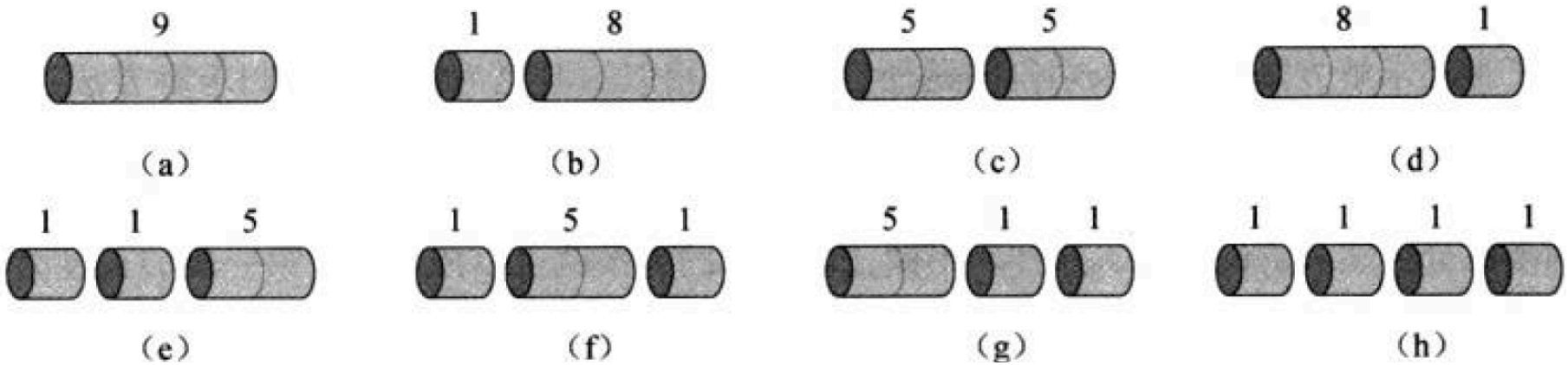
- Serling公司购买长钢条，将其切割为短钢条出售。切割工序本身没有成本支出。公司管理层希望知道最佳的切割方案。假定我们知道Serling公司**出售一段长为*i*英寸的钢条的价格为p_i(*i*=1,2,…，单位为美元)**。钢条的长度均为整英寸。图15-1给出了一个价格表的样例。
- 钢条切割问题是这样的：给定一段长度为*n*英寸的钢条和一个价格表p_i(*i*=1,2,…n)，求切割钢条方案，使得**销售收益r_n**最大。
- 注意，如果长度为*n*英寸的钢条的价格p_n足够大，最优解可能就是完全不需要切割

长度*i*	1	2	3	4	5	6	7	8	9	10
价格p_i	1	5	8	9	10	17	17	20	24	30

图 15-1　钢条价格表样例。每段长度为 *i* 英寸的钢条为公司带来 p_i 美元的收益

习题1：钢条切割问题-切割方案举例

- 4英寸钢条（n=4）的八种切割方案：



长度i	1	2	3	4	5	6	7	8	9	10
价格p_i	1	5	8	9	10	17	17	20	24	30

图 15-1　钢条价格表样例。每段长度为 i 英寸的钢条为公司带来 p_i 美元的收益

习题1：钢条切割问题

- 长度为n英寸的钢条共有2^{n-1}种不同的切割方案，因为在距离钢条左端$i(i=1,2,\ldots n)$英寸处，总是可以选择切割或不切割。
- 问题建模：

如果一个最优解将钢条切割为 k 段(对某个 $1\leqslant k\leqslant n$)，那么最优切割方案

$$n = i_1 + i_2 + \cdots + i_k$$

将钢条切割为长度分别为 i_1，i_2，…，i_k 的小段，得到最大收益

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

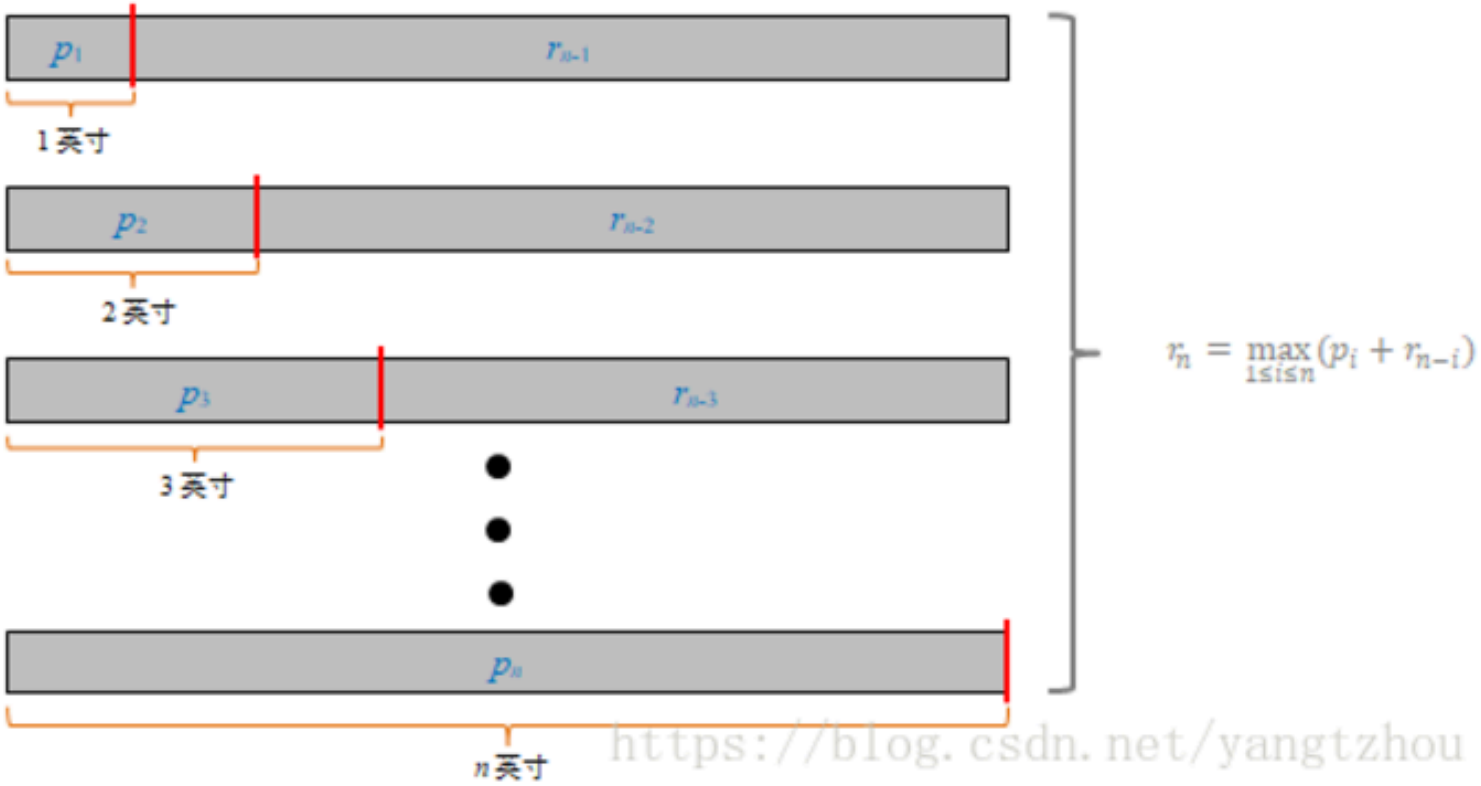
更一般地，对于 $r_n(n\geqslant 1)$，我们可以用更短的钢条的最优切割收益来描述它：

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

- **步骤一**：刻画一个最优解的结构特征：寻找最优子结构（*问题的一个最优解包含了子问题的最优解。*）

习题1：钢条切割问题

- 将长度为 n 的钢条切割为两段：长度为 i 的一段和长度为 $n-i$ 的另一段（其中 $1 \le i \le n$）。



https://blog.csdn.net/yangtzhou

习题1：钢条切割问题

- 最优子结构：问题可以分解为规模更小的子问题；问题的最优解依赖于子问题的最优解，并且这些子问题可以独立求解

递归定义：

$$r(n) = \max_{1 \le i \le n} \{p[i] + r(n-i)\}$$

其中：

- $p[i]$：长度为 i 的钢条收益；
- $r(n-i)$：剩余长度为 $n-i$ 的钢条收益的最优解。

习题1：钢条切割问题

最优子结构证明

要证明钢条切割问题具有最优子结构性质，需要说明以下两点：

1. **问题的最优解包含子问题的最优解：**
   若长度为 n 的钢条的最优切割方案将钢条分为长度为 i 和 $n-i$ 的两段，则该方案的总收益为：

   $$r(n) = p[i] + r(n-i)$$

   假设子问题 $r(n-i)$ 的解不是最优的，那么我们可以用更优的解替换它，从而得到更大的 $r(n)$。但这与 $r(n)$ 是最优解的假设矛盾。因此，子问题 $r(n-i)$ 必须是最优解。

2. **子问题独立性：**
   钢条的收益仅依赖于其长度，与其他部分无关。因此，求解长度为 $n-i$ 的钢条收益时，不影响长度为 i 的收益。

因此，钢条切割问题的最优解可以通过其子问题的最优解组合而成，这正是最优子结构性质的定义。

习题1：钢条切割问题

- 证明最优子结构：
- 如果长度为*n*的钢条的最优切割方案将其分割为长度为*i*和*n*–*i*的两部分，则**子问题 r(*n*–*i*)的解是最优的**

假设

- $r(n) = \max_{1 \le i \le n}\{p[i] + r(n-i)\}$，其中 $r(n)$ 表示长度为 n 的钢条的最优收益。
- 最优切割方案选择了某个 i^*，使得：

$$r(n) = p[i^*] + r(n - i^*)$$

反证法

假设子问题 $r(n-i^*)$ 的解不是最优的，即存在另一种方案，使得收益更高，记为 $r'(n-i^*)$，且 $r'(n-i^*) > r(n-i^*)$。

1. 如果用 $r'(n-i^*)$ 替代 $r(n-i^*)$，则总收益变为：

$$r'(n) = p[i^*] + r'(n - i^*)$$

2. 因为 $r'(n-i^*) > r(n-i^*)$，所以：

$$r'(n) > p[i^*] + r(n - i^*) = r(n)$$

3. 这表明存在更优的总收益 $r'(n)$，与 $r(n)$ 是最优解的假设矛盾。

习题1：钢条切割问题

■ 步骤二：递归地定义最优解的值

- 递归公式：

$$r_n = \max_{1 \leqslant i \leqslant n}(p_i + r_{n-i})$$

- 递归过程：
  - *p*:价格数组
  - *n*：钢条长度
  - *q*：最大收益

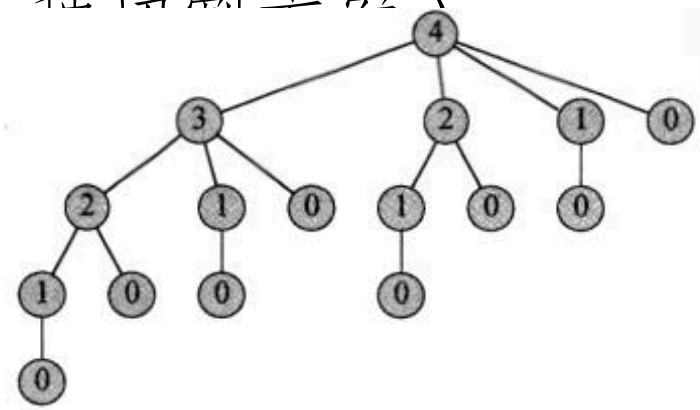
```
CUT-ROD(p,n)
if n == 0
    return 0
q=-∞
for i = 1 to n
    q=max(q,p[i]+CUT-ROD(p,n-i))
return q
```

- 运行时间：
- 每当*n*增大1，程序运行时间差不多会增加1倍！

习题1：钢条切割问题

■ 步骤二：递归地定义最优解的值

- 一种直接的自顶向下的递归方法
- 效率分析
  - 反复求解相同的子问题
  - 举例：n=4（从根节点到叶节点的一条路径对应长度为n的钢条的一种切割方案）



这棵递归调用树显示了 $n=4$ 时，CUT-ROD(p，n)的递归调用过程。每个结点的标号为对应子问题的规模 n，因此，从父结点 s 到子结点 t 的边表示从钢条左端切下长度为 $s-t$ 的一段，然后继续递归求解剩余的规模为 t 的子问题。从根结点到叶结点的一条路径对应长度为 n 的钢条的 2^{n-1} 种切割方案之一。一般来说，这棵递归调用树共有 2^n 个结点，其中有 2^{n-1} 个叶结点

习题1：钢条切割问题

■ 步骤三：计算最优解的值

- 基本思路：仔细安排求解顺序，对每个子问题只求解一次，并将结果保存下来。
  - 如果随后再次需要此子问题的解，只需查找保存的结果。
  - 常用的算法设计思想：付出额外的内存空间来节省时间（时空权衡）。在本例中，时间上的节省是巨大的。

- 问题：如何安排子问题的求解顺序？
  - （1）带备忘的自顶向下法
  - （2）自底向上法

习题1：钢条切割问题

- 动态规划有两种等价的实现方法。

（1）带备忘的自顶向下法

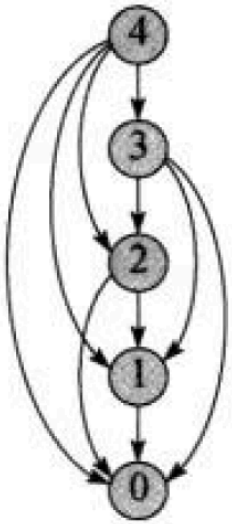
- 仍按自然的递归形式编写过程，但过程会保存每个子问题的解（通常保存在一个数组或散列表中）。当需要一个子问题的解时，过程首先检查是否已经保存过此解。如果是，则直接返回保存的值，从而节省了计算时间；否则，按通常方式计算这个子问题。
- **带备忘的：因为记住了之前已经计算出的结果**

（2）自底向上法

- 一般需要恰当定义子问题“规模”的概念，使得任何子问题的求解都只依赖于“更小的”子问题的求解。因此，我们可以将子问题按照规模顺序，由小至大顺序进行求解。当求解某个子问题时，它所依赖的那些更小的子问题都已求解完毕，结果已经保存。每个子问题只需求解一次，当我们求解它时，它的所有前提子问题都已求解完成。
- **逆拓扑序（反序的拓扑序）：对于任何子问题，直至它依赖的所有子问题都已求解完成，才会求解它。**

习题1：钢条切割问题

- 子问题图：子问题及子问题之间的依赖关系
  - 逆拓扑序



$n=4$ 时，钢条切割问题的子问题图。顶点的标号给出了子问题的规模。有向边(x, y)表示当求解子问题x时需要子问题y的解。此图实际上是图 15-3 中递归调用树的简化版——树中标号相同的结点收缩为图中的单一顶点，所有边均从父结点指向子结点

习题1：钢条切割问题

- 问题：两种实现方式的伪代码？

习题1：钢条切割问题

- 自顶向下

```
MEMOIZED-CUT-ROD(p,n)
let r[0..n]be a new array
for i = 0 to n
    r[i]=-∞
return MEMOIZED-CUT-ROD-AUX(p,n,r)
```

```
MEMOIZED-CUT-ROD-AUX(p,n,r)
if r[n]≥0
    return r[n]
if n == 0
    q = 0
else q=-∞
    for i = 1 to n
        q=max(q,p[i]+MEMOIZED-CUT-ROD-AUX(p,n-i,r))
r[n]=q
return q
```

习题1：钢条切割问题

- 自底向上

```
BOTTOM-UP-CUT-ROD(p,n)
let r[0..n]be a new array
r[0]=0
for j = 1 to n
    q=-∞
    for i = 1 to j
        q=max(q,p[i]+r[j-i])
    r[j]=q
return r[n]
```

习题1：钢条切割问题

■ **步骤四：利用计算出的信息构造解**

- 前面PPT中返回了最优解的收益值，但并未返回解本身：一个长度列表，给出切割后每段钢条的长度
- 扩展动态规划算法：不仅保存最优收益值，还保存对应的切割方案

- 算法伪代码?

习题1：钢条切割问题

- 与BOTTOM-UP-CUT-ROT的区别：在第一行中创建了数组*s*，并在求解规模为*j*的子问题时将第一段钢条的最优切割长度*i*保存在*s*[*j*]中（第8行）

```
EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
1   let r[0..n]and s[0..n]be new arrays
2   r[0]=0
3   for j = 1 to n
4       q=-∞
5       for i = 1 to j
6           if q < p[i]+r[j-i]
7               q=p[i]+r[j-i]
8               s[j]=i
9       r[j]=q
10  return r and s
```

- 输出长度为*n*的钢条的完整的最优方案：

```
PRINT-CUT-ROD-SOLUTION(p,n)
(r,s)=EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
while n>0
    print s[n]
    n=n-s[n]
```

习题1：钢条切割问题

- 切割方案

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

动态规划(Dynamic programming)

- 设计动态规划算法的步骤：
  - 1. 刻画一个最优解的结构特征：寻找最优子结构，从子问题的最优解构造出原问题的最优解
  - 2. 递归地定义最优解的值：用子问题的最优解来递归地定义原问题最优解的代价
  - 3. 计算最优解的值，通常采用自底向上方法
  - 4.利用计算出的信息构造一个最优解

习题2：矩阵链乘法问题

◆ 问题描述：

- 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, $(i = 1,\ldots, n-1)$
- 希望计算这n个矩阵的连乘积$A_1A_2\ldots A_n$
  - A_i与A_{i+1}可乘，条件：$col(A_i)=row(A_{i+1})$

- 由于矩阵乘法满足结合律，因此任何加括号的方法都会得到相同的计算结果
  - “完全括号化”：它是单一矩阵或者是两个完全括号化的矩阵乘积链的积

- 问题：设矩阵A_i的规模为$p_{i-1} * p_i$,求完全括号化方案，使得计算乘积所需标量乘法次数最少

习题2：矩阵链乘法问题

1.问题分析：

（1）两个矩阵的相乘次数：

$p*q*r$

（2）举例分析：

10*100, 100*5, 5*50

方案一：

10*100*5+10*5*50=7500

方案二：

10*100*50+100*5*50=75000

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
return C
```

习题2：矩阵链乘法问题

1.问题分析：

（3）直接的递归方法

$$P(n)=\begin{cases}1 & \text{如果 } n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{如果 } n\geqslant 2\end{cases}$$

习题2：矩阵链乘法问题

1.问题分析：

(4) 分析最优解结构

将 $A_1A_2\ldots A_n$=记为$A_{1..n}$,

设在第k个位置上断开($1\le k\le n$),

即对应计算为：（($A_1\ldots A_k$)（$A_{k+1}\ldots A_n$））

计算量：包括3部分：

($A_1\ldots A_k$)，（$A_{k+1}\ldots A_n$）和（$\underline{(A_1\ldots A_k)}$ $\underline{(A_{k+1}\ldots A_n)}$）

- 关键特征：计算$A_{1..n}$的一个最优次序所包含的$A_{1..k}$次序也是最优的。

  由此可知，矩阵链乘积的计算次序问题的最优解包含着其子问题的最优解----称这种性质为最优子结构性质。

  这是可用动态规划算法的基本要素

- 如何证明具有最优子结构？-》反证法

习题2：矩阵链乘法问题

2. 建立递归关系

- 假设计算$A_{i..j}$,所需要的最少数乘次数为$m(i, j)$，则原问题的最优值为$m(1, n)$。
  - 当$i=j$时，无需计算，$m(i, j)=m(i, i)=0$
  - 当$i<j$时，可用最优子结构性质来计算$m(i, j)$:
    - 若在k处断开，$i \le k<j$，则

$$m(i, j)=m(i, k) + m(k+1, j) + p_{i-1}p_kp_j$$

  - 由于不知k的位置，需要在$j-i$个位置上选择使得$m(i, j)$最小的k。

即可定义如下：

$$m(i, j)=\begin{cases} 0 & i=j\text{时} \\ min(m(i, k)+ m(k+1, j) + p_{i-1}p_kp_j) & i<j\text{时} \end{cases}$$

若将$m(i, j)$对应的k值记录在$s(i, j)$中，则相应的最优解便可构造出来

习题2：矩阵链乘法问题

3. 计算最优解（自底向上）

- 根据上述$m(i, j)$的计算公式，容易写出递归程序计算$m(1, n)$。
  - 不同的子问题的个数是$O(n^2)$.
  - 因此，许多计算是重复的。

- 为此，可采用自底向上的方法计算，
  计算过程中，保存已经计算过的子问题的答案，
  以便后面更高层的计算直接引用。

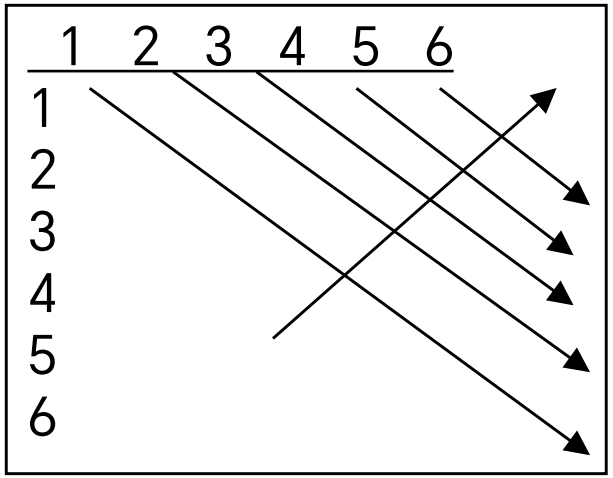
习题2：矩阵链乘法问题

例如，矩阵$A_1A_2A_3A_4A_5A_6$的维数分别如下：

A_1	A_2	A_3	A_4	A_5	A_6
30*35	35*15	15*5	5*10	10*20	20*25

P=（30，35，15，5，10，20，25）

为了计算矩阵连乘积$A_1A_2A_3A_4A_5A_6$的最优次序，计算过程如下：



	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

m[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

s[i, j]

习题2：矩阵链乘法问题

- 自下而上、自左至右的顺序计算所有行
- 当计算表项$m[i, j]$时，会用到乘积$p_{i-1}p_kp_j(k=i, i+1,\ldots,j-1)$，以及$m[i, j]$西南方向和东南方向的所有表项

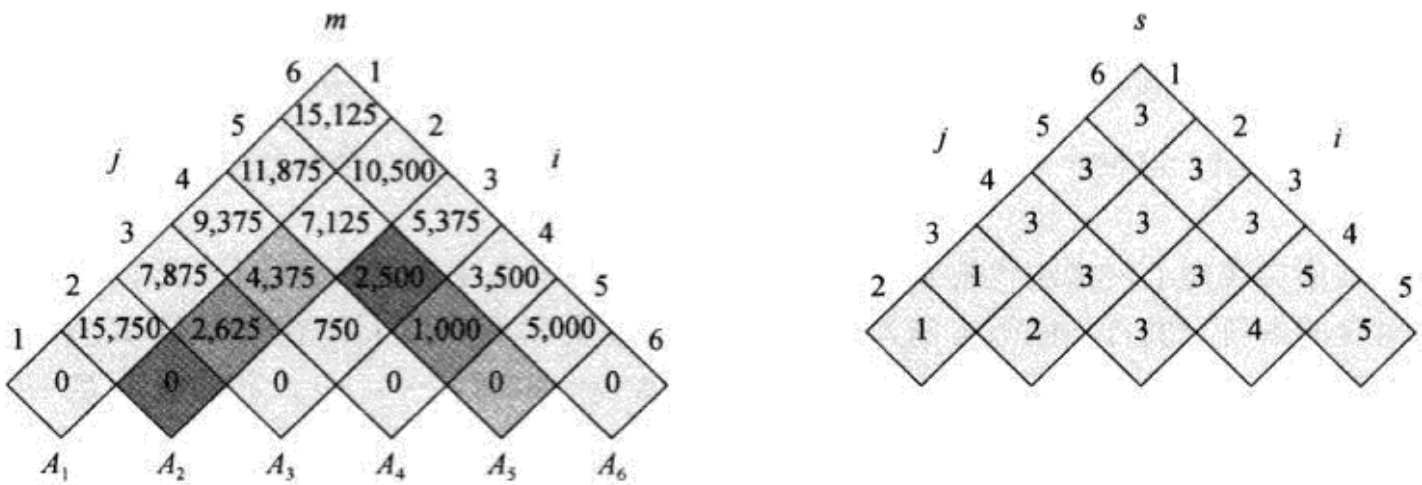


图 15-5　当 $n=6$ 和矩阵规模如下表时，MATRIX-CHAIN-ORDER 计算出的 m 表和 s 表。

矩阵	A_1	A_2	A_3	A_4	A_5	A_6
规模	30×35	35×15	15×5	5×10	10×20	20×25

习题2：矩阵链乘法问题

由此可得计算过程如下：

- 先计算出$m(i, i)=0$,　$i=1,2,\ldots,n$
- 计算$m(i, i+1)$,　$i=1,2,\ldots,n-1$

  矩阵链长度为2
- 计算$m(i, i+2)$,　i=1,2,…,n-2

  矩阵链长度为3
- …
- 在计算$m(i, j)$时，

  只用到已经计算出的$m(i, k)$,$m(k+1, j)$

矩阵 列数序列

$P=\{p_0, p_1, p_2,\ldots, p_n\}$

```
void matrix_chain_order(P)
{
n=len(p)-1;
for (i=1; i<=n;i++) m[i, i]=0;
for (L=2; L<=n; L++)
for (i=1; i<=n-L+1;i++)
  {
     j=i+L-1;
     m[i, j]=∞;
    for (k=i; k<=j-1; k++)
      {
        q=m[i, k]+m[k+1, j]+ p_{i-1}p_k p_j;
        if (q<m[i, j])
        { m[i, j]=q; s[i, j]=k; }
      }
}
```

习题2：矩阵链乘法问题

- 例如，计算$m(2, 5)$时，依据计算式有

$$m(2, 5)=\min\begin{cases} m(2,2) + m(3,5) + p_1p_2p_5=0+2500+35*15*20=13000 \\ m(2,3) + m(4,5)+ p_1p_3p_5=2625+1000+35*5*20=7125 \\ m(2,4) + m(5,5)+p_1p_4p_5=4375+0+35*10*20=11375 \end{cases}$$

$=7125,$

$k=3,\ s(2,5)=3$

	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

A_1	A_2	A_3	A_4	A_5	A_6
30*35	35*15	15*5	5*10	10*20	20*25

习题2：矩阵链乘法问题

4. 构造最优解

- 算法matrix_chain_order(P)给出了最优值，但未给出最优解。下面讨论最优解的求解。
- 事实上，$s[i, j]$中的k值已经给出了信息。由此可以构造出求解序列：例如，前例1..6的序列
- P=（30，35，15，5，10，20，25）

```
void   matrix_chain_multiply(A, s, i, j)    // 矩阵相连乘
  {
    if(i == j) return(A_i)
    else{
      x=matrix_chain_multiply(A, s, i, s[i, j]);
      y=matrix_chain_multiply(A, s, s[i, j]+1, j);
      return matrix_multiply(x, y);
    }
}
```

	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

$s[i, j]$

习题2：矩阵链乘法问题

思路扩展：矩阵链乘法问题可以使用哪些算法求解（不考虑时间复杂度）？

s[i,j]

贪心法求解思路?

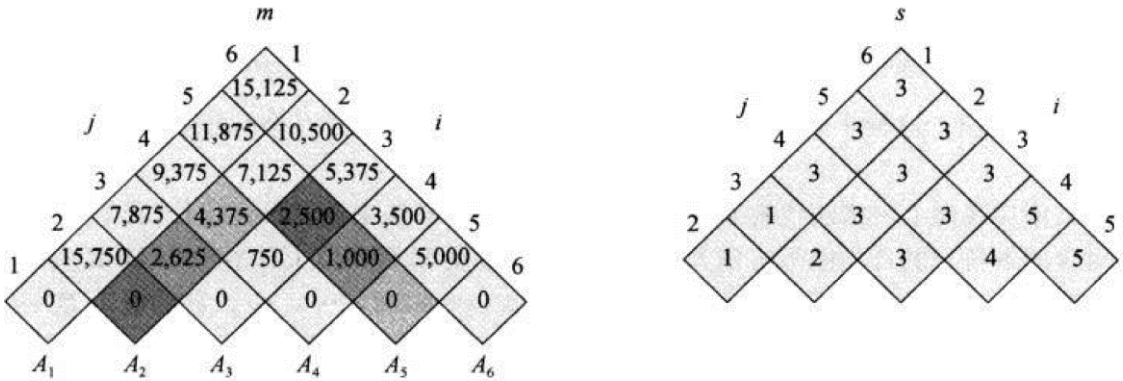


图 15-5 当 $n=6$ 和矩阵规模如下表时，MATRIX-CHAIN-ORDER 计算出的 m 表和 s 表。

矩阵	A_1	A_2	A_3	A_4	A_5	A_6
规模	30×35	35×15	15×5	5×10	10×20	20×25

- 计算$m(2, 5)$时，动态规划计算式：

$$m(2,5)=\min\begin{cases} m(2,2)+m(3,5)+p_1p_2p_5 = 0+2500+35*15*20 = 13000 \\ m(2,3)+m(4,5)+p_1p_3p_5 = 2625+1000+35*5*20 = 7125 \\ m(2,4)+m(5,5)+p_1p_4p_5 = 4375+0+35*10*20 = 11375 \end{cases}$$

$=7125$,

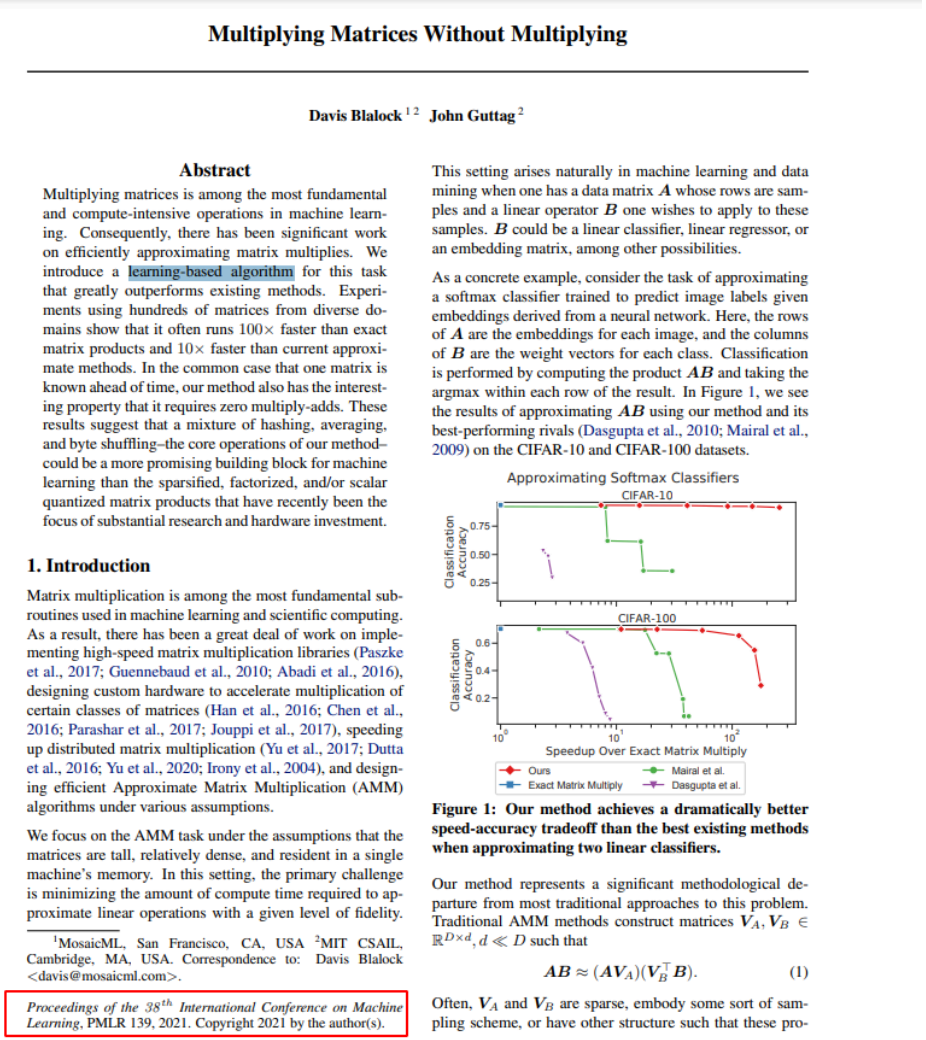
1-6最优方案：$((A_1,(A_2,A_3))((A_4,A_5)A_6))$

习题2：矩阵链乘法问题

◆思路扩展：矩阵链乘法问题可以使用哪些算法求解？

- 暴力解法
- 分治法
- Stassen算法
- 贪心法
- 动态规划
- 基于学习的方法？

◆为什么不用某些算法求解？-》算法分析（时间复杂度、空间复杂度）



Multiplying Matrices Without Multiplying

Davis Blalock [1 2] John Guttag [2]

Abstract

Multiplying matrices is among the most fundamental and compute-intensive operations in machine learning. Consequently, there has been significant work on efficiently approximating matrix multiplies. We introduce a learning-based algorithm for this task that greatly outperforms existing methods. Experiments using hundreds of matrices from diverse domains show that it often runs 100× faster than exact matrix products and 10× faster than current approximate methods. In the common case that one matrix is known ahead of time, our method also has the interesting property that it requires zero multiply-adds. These results suggest that a mixture of hashing, averaging, and byte shuffling–the core operations of our method–could be a more promising building block for machine learning than the sparsified, factorized, and/or scalar quantized matrix products that have recently been the focus of substantial research and hardware investment.

1. Introduction

Matrix multiplication is among the most fundamental subroutines used in machine learning and scientific computing. As a result, there has been a great deal of work on implementing high-speed matrix multiplication libraries (Paszke et al., 2017; Guennebaud et al., 2010; Abadi et al., 2016), designing custom hardware to accelerate multiplication of certain classes of matrices (Han et al., 2016; Chen et al., 2016; Parashar et al., 2017; Jouppi et al., 2017), speeding up distributed matrix multiplication (Yu et al., 2017; Dutta et al., 2016; Yu et al., 2020; Irony et al., 2004), and designing efficient Approximate Matrix Multiplication (AMM) algorithms under various assumptions.

We focus on the AMM task under the assumptions that the matrices are tall, relatively dense, and resident in a single machine's memory. In this setting, the primary challenge is minimizing the amount of compute time required to approximate linear operations with a given level of fidelity.

[1]MosaicML, San Francisco, CA, USA [2]MIT CSAIL, Cambridge, MA, USA. Correspondence to: Davis Blalock <davis@mosaicml.com>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

This setting arises naturally in machine learning and data mining when one has a data matrix $\boldsymbol{A}$ whose rows are samples and a linear operator $\boldsymbol{B}$ one wishes to apply to these samples. $\boldsymbol{B}$ could be a linear classifier, linear regressor, or an embedding matrix, among other possibilities.

As a concrete example, consider the task of approximating a softmax classifier trained to predict image labels given embeddings derived from a neural network. Here, the rows of $\boldsymbol{A}$ are the embeddings for each image, and the columns of $\boldsymbol{B}$ are the weight vectors for each class. Classification is performed by computing the product $\boldsymbol{AB}$ and taking the argmax within each row of the result. In Figure 1, we see the results of approximating $\boldsymbol{AB}$ using our method and its best-performing rivals (Dasgupta et al., 2010; Mairal et al., 2009) on the CIFAR-10 and CIFAR-100 datasets.

Figure 1: Our method achieves a dramatically better speed-accuracy tradeoff than the best existing methods when approximating two linear classifiers.

Our method represents a significant methodological departure from most traditional approaches to this problem. Traditional AMM methods construct matrices $\boldsymbol{V}_A, \boldsymbol{V}_B \in \mathbb{R}^{D \times d}, d \ll D$ such that

$$\boldsymbol{AB} \approx (\boldsymbol{AV}_A)(\boldsymbol{V}_B^\top \boldsymbol{B}). \quad (1)$$

Often, $\boldsymbol{V}_A$ and $\boldsymbol{V}_B$ are sparse, embody some sort of sampling scheme, or have other structure such that these pro-

A Dynamic Progra based Energy-Ef

Shervin Hajiamini, Behro

Abstract—Per-core Dynamic Voltage and Fre efficiency in multicore systems. Heuristic DVF Programming (DP) methods solve smaller sul the cost of overhead delays. We propose an e Product (EDP) as objective function to predict consumption and execution time. Experiment and provides near-optimal solutions when ma penalty. In fact, across several benchmarks, c heuristic methods. Furthermore, using a Pare our framework's energy-time solution is on av dynamic programming solution is 3 to 18% cl

Index Terms—Dynamic voltage and frequenc frontier.

9 CONCLUSION AND FUTURE WORK

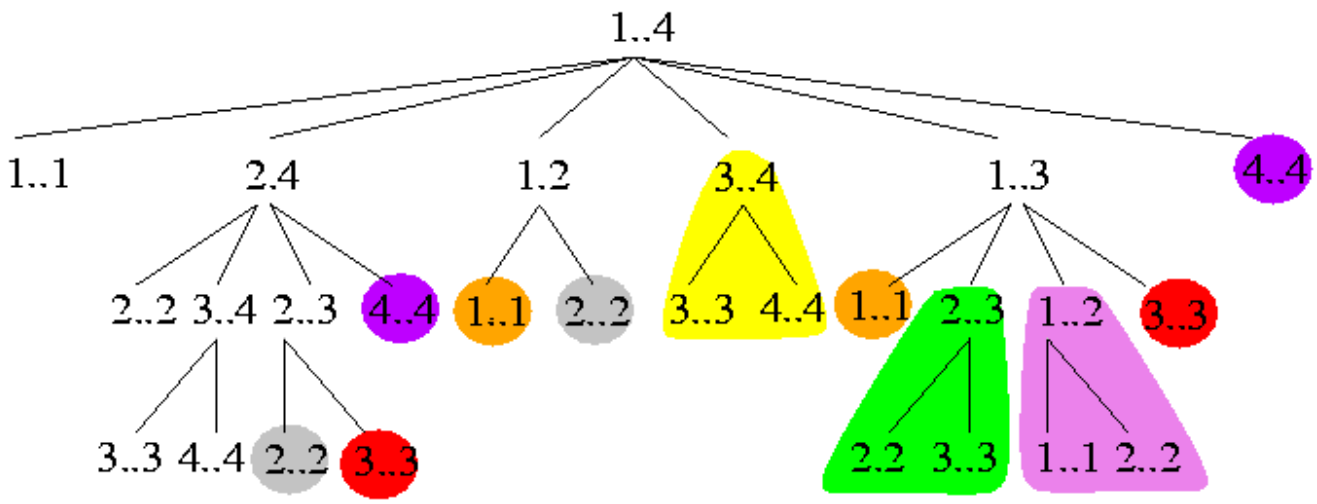
This paper proposes a Dynamic Programming (DP) framework, based on the Viterbi algorithm, to achieve fine-grain, per core energy-time tradeoff analysis. This technique globally optimizes per-core V/F levels by minimizing the cores' energy consumptions and execution times. The performance of the framework is compared to faster version of the VDP algorithm (Greedy) and two other heuristic algorithms (Feedback and Ondemand). The EDP performance results show that, on the average, our proposed VDP algorithm outperforms Greedy by 12%, Feedback by 25% and Ondemand by 75%.

Furthermore, the results show that the VDP algorithm performs the best when maximizing energy saving is as, or more, important than minimizing the execution time penalty. Considering the best possible optimal solution for each algorithm, the results show that VDP, compared to the other heuristics, has the closest performance to its corresponding optimal solution.

As discussed in Section 4.2, the algorithms presented in this paper perform compile-time DVFS and are usable for applications with a specific execution model (Fig. 1). Furthermore, these algorithms are applied on a system with a moderate number of homogenous cores as explained in Section 7.2. To further extend these algorithms, future work will address the following points: 1) Studying applications with different execution models, 2) Analyzing the scalability and energy efficiency of the algorithms with respect to larger system sizes, 3) Integrating the algorithms with OS kernels to perform runtime DVFS, and 4) Evaluating the performance of the algorithms on systems with heterogeneous cores.

动态规划(DP)与分而治之

- 两者都是合并子问题的解来解决问题
- 分而治之算法
  - 将问题划分为独立的子问题，递归的解决子问题，然后将他们的解组合得到原始问题的解。
  - 如果相同的子问题出现多于一次，算法效率不高。
- 动态规划 (DP)
  - 适合子问题重叠的情况
  - D



动态规划(DP)

- 典型应用为**最优问题**.
- 通用方法
  - 计算所有子问题的解。
  - 利用小的子问题来解决大的子问题。
  - 子问题的计算基于以前计算的更小的子问题的结果。
  - 将子问题的结果存储在表中，不再重新计算。
- 开发 DP
  - 研究最优方案的结构
  - 递归定义最优方案的值
  - 从底向上计算最优方案的值
  - 从计算的结果创建最优方案（如果仅仅需要最优值，此步可以忽略）。

什么时候使用动态规划(DP)

- DP 通过存储部分结果来进行高效的递归计算　仅仅在部分结果的数目比较小的时候。
- **无望的情况:** n个元素集合的排列组合$n!$，n个元素的2^n个子集，等等。
- **有希望的情况:**
  - n个字符的字符串的连续子串
  - 二叉搜索树的可能子树，等等。
- **DP 的最佳应用是线性有序但是却不能排序的对象。**
  - 矩阵链，字符串中的字符，多边形边界周围的点，圆上的点，搜索树中叶子从左到右的顺序，等等。
  - 对象从左到右有序

DP的两种方法

- 从下向上迭代方法
  - 从递归的分而治之算法开始
  - 找到子问题之间的依赖关系（计算子问题需要什么样的解）
  - 按照正确的顺序解决子问题
- 从上到下的递归方法(记忆)
  - 从递归的分而治之算法开始
  - 保持原始算法从上到下的方法
  - 将子问题的解保存在表中(可能需要很多存储空间）
  - 只有当子问题的解在表中没有的时候才进行递归
- 如果所有的子问题都必须计算至少一次，从下向上的方法更好，因为减少了递归和维护表的负担。
- 如果很多子问题不需要计算，从上到下的DP方法更好，因为只有用到的项被计算。

动态规划原理

- 动态规划算法的有效性依赖于两个重要性质：
  - 子问题重叠性质
  - 最优子结构性质

这是采用动态规划算法的基本要素。

（1）重叠子问题

- 子问题的重叠性质：
  - 每次产生的子问题并不总是新的问题，有些问题被反复计算。
  - 因此，采用动态规划法只计算每个子问题一次，并放到表中，需要时再查表即可。
  - 例如，矩阵连乘计算问题中，n^2数量级的求解即可。

动态规划原理

（2）最优子结构性质

- 当问题的最优解包含其子问题的最优解时，称该问题具有最优子结构性质。
- 提供了可用动态规划算法求解的线索。
  - 例：矩阵连乘问题，分解为子问题（加括号方式求解）
- 可以采用自底向上的方式求解，
- 并且可以在相对较小的子问题空间中考虑问题。

习题3：最长公共子序列-示例

- 最长公共子序列 (LCS)
- 给定两个序列 $x[1 \ldots m]$ 和 $y[1 \ldots n]$,找出他们两者一个最长公共子序列。
- 例如：
- A = "HelloWorld"
- B = "loop"
- 则A与B的最长公共子序列为 "loo",返回的长度为3

习题3：最长公共子序列-示例

- 例子: 最长公共子序列 (LCS)
- 给定两个序列 $x[1 \ldots m]$ 和 $y[1 \ldots n]$,找出他们两者一个最长公共子序列。

x:	A	B	C	B	D	A	B
y:	B	D	C	A	B	A	

习题3：最长公共子序列-示例

- 给定两个序列 $x[1..m]$ 和 $y[1..n]$,找出他们两者一个最长公共子序列。

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

函数标记，并不是函数

习题3：最长公共子序列-野蛮算法

- 检查$x[1..m]$ 的每个子序列 。看它是否为$y[1..n]$的一个子序列。

习题3：最长公共子序列-野蛮算法

- 检查$x[1 . . m]$ 的每个子序列 。看它是否为$y[1 . . n]$的一个子序列。

分析

- 对每个子序列，检查 $= O(n)$ 时间
- 序列x有2^m个子序列(每个长度为 m 的位矢量对应一个x的子序列).

最坏情况下的运行时间 $= O(n2^m)$

$=$ 指数时间.

习题3：最长公共子序列-向更好的算法努力

- 简化:
  - 观察 一个最长公共子序列的长度。
  - 扩展算法找到 LCS 。

习题3：最长公共子序列-向更好的算法努力

- **简化:**
  - 观察 一个最长公共子序列的*长度*
  - 扩展算法找到 LCS
- **表示法:** $|s|$表示序列的s长度
- **策略:** 考虑 x 和 y的*前缀*
  - 定义 $c[i, j] = |LCS(x[1..i], y[1..j])|$
  - 那么 $c[m, n] = |LCS(x, y)|$

动态规划标志#1

最优子结构

问题的一个最优解包含了子问题的最优解。

习题3：最长公共子序列：LCS的最优子结构

- 定理（LCS的最优子结构）：令$X=x[1..m]$和$Y=y[1..n]$为两个序列，$Z=z[1..k]$为X和Y的任意LCS。
- 1. 如果$x[m]=y[n]$，则$z[k]=x[m]=y[n]$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS
- 2.如果$x[m] \neq y[n]$，则$z[k] \neq x[m]$意味着Z是X_{m-1}和Y的一个LCS
- 3.如果$x[m] \neq y[n]$，则$z[k] \neq y[n]$意味着Z是X和Y_{n-1}的一个LCS

习题3：最长公共子序列-递归公式

证明思路：反证法！

证明：

- 情况(1) 如果$x[m] = y[n]$，则$z[k]=x[m]= y[n]$且Z_{k-1}是X_{m-1}和Y_{n-1}的一个LCS

- 如果$z[k]\neq x[i]$，那么,可以将$x[m] = y[n]$追加到Z的末尾，扩展 Z得到长度为$k+1$的公共子序列(矛盾)；
- 这样, Z_{k-1}是X_{m-1}和Y_{n-1}的一个公共子序列。
- 只需证：$z[1 \ldots k-1]$是$x[1 \ldots m-1]$ 和 $y[1 \ldots n-1]$的LCS。（待证）

习题3：最长公共子序列-递归公式

- 证明：$z[1..k-1]$是$x[1..m-1]$ 和 $y[1..n-1]$的 LCS。

- 假设w是$x[1..m-1]$ 和$y[1..n-1]$一个更长的 LCS，即 $|w| > k-1$. 那么，***剪贴并且拷贝*** $w \| z[k]$ (w与$z[k]$ 连接) 是$x[1..m]$ 和 $y[1..n]$ 的公共子序列，同时$|w \| z[k]| > k$。

自相矛盾，由此声明得证。

其他情况类似。

习题3：最长公共子序列-递归公式

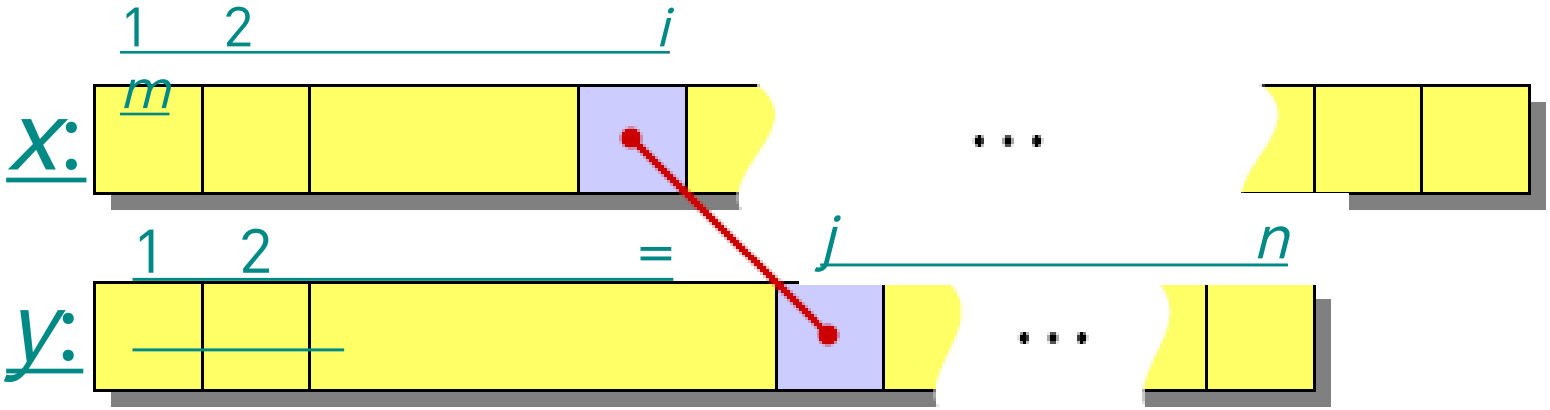
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{如果 } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{其他.} \end{cases}$$

- 如果$x[m] = y[n]$，我们应该求解Xm-1和Yn-1的一个LCS
- 如果$x[m] \neq y[n]$，我们必须求解两个子问题：
  - ✓ 求Xm-1和Yn的一个LCS
  - ✓ 求Xm和Yn-1的一个LCS

习题3：最长公共子序列-递归公式

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{如果 } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{其他.} \end{cases}$$

情况 $x[i] = y[j]$



习题3： LCS的递归算法

LCS(x, y, i, j)

if $x[i] = y[j]$

then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$

else $c[i, j] \leftarrow \max\{$ LCS$(x, y, i-1, j)$,

LCS$(x, y, i, j-1)\}$

习题3： LCS的递归算法

LCS(x, y, i, j)

if $x[i] = y[j]$

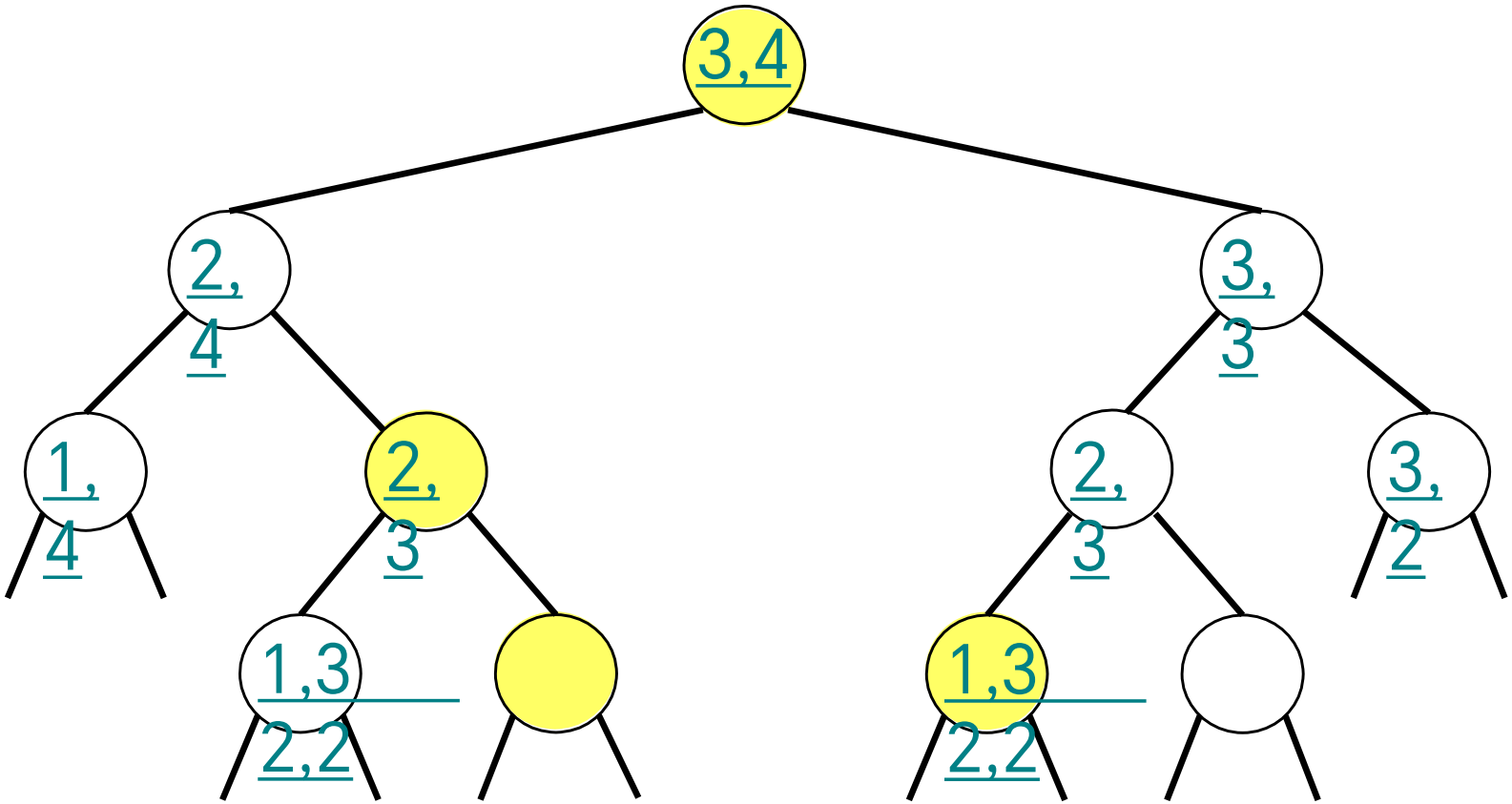
then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$

else $c[i, j] \leftarrow \max\{$ LCS$(x, y, i-1, j)$,

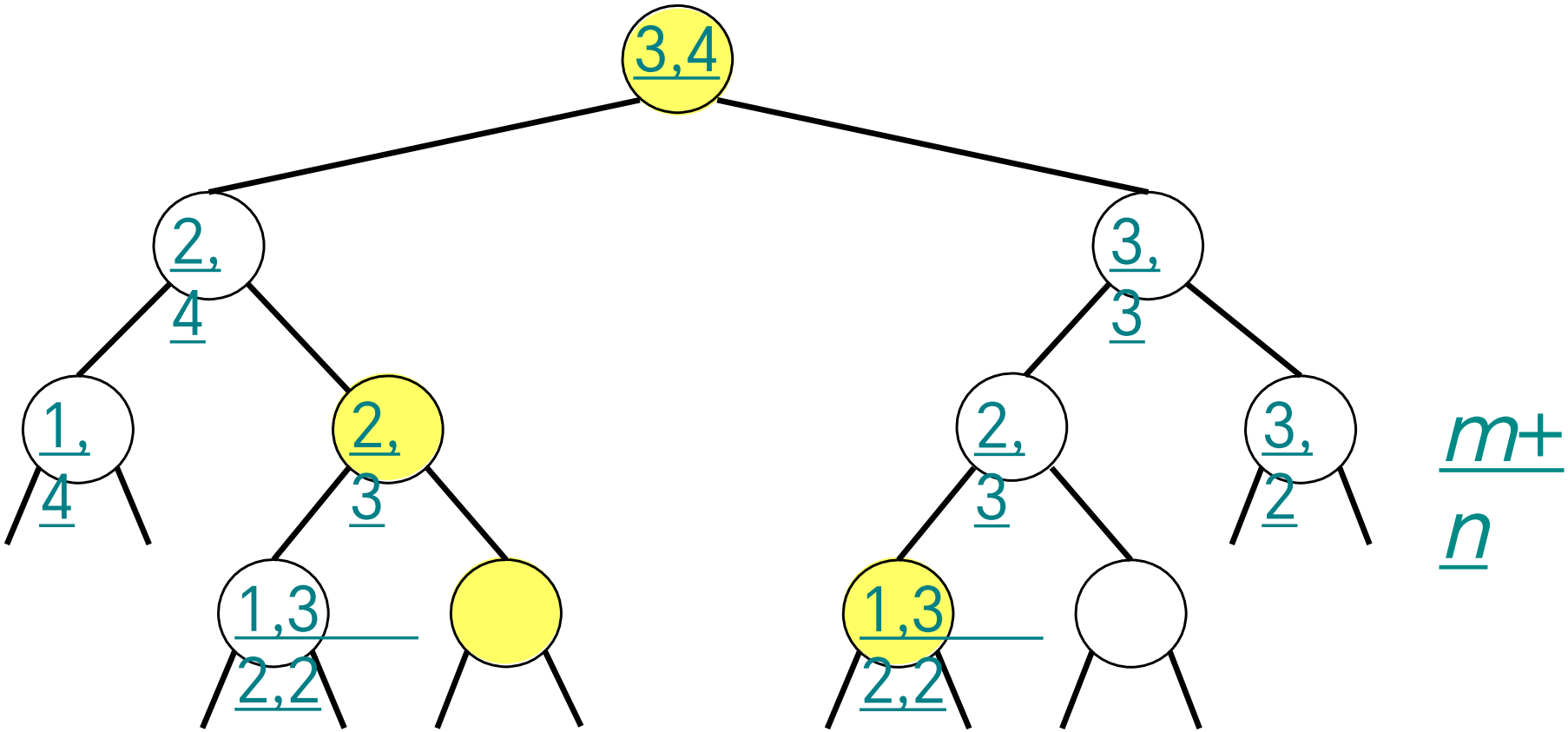
LCS$(x, y, i, j-1)\}$

最坏情况: $x[i] \neq y[j]$, 这种情况下算法要计算两个子问题,每个子问题只有一个参数递减了。

递归树



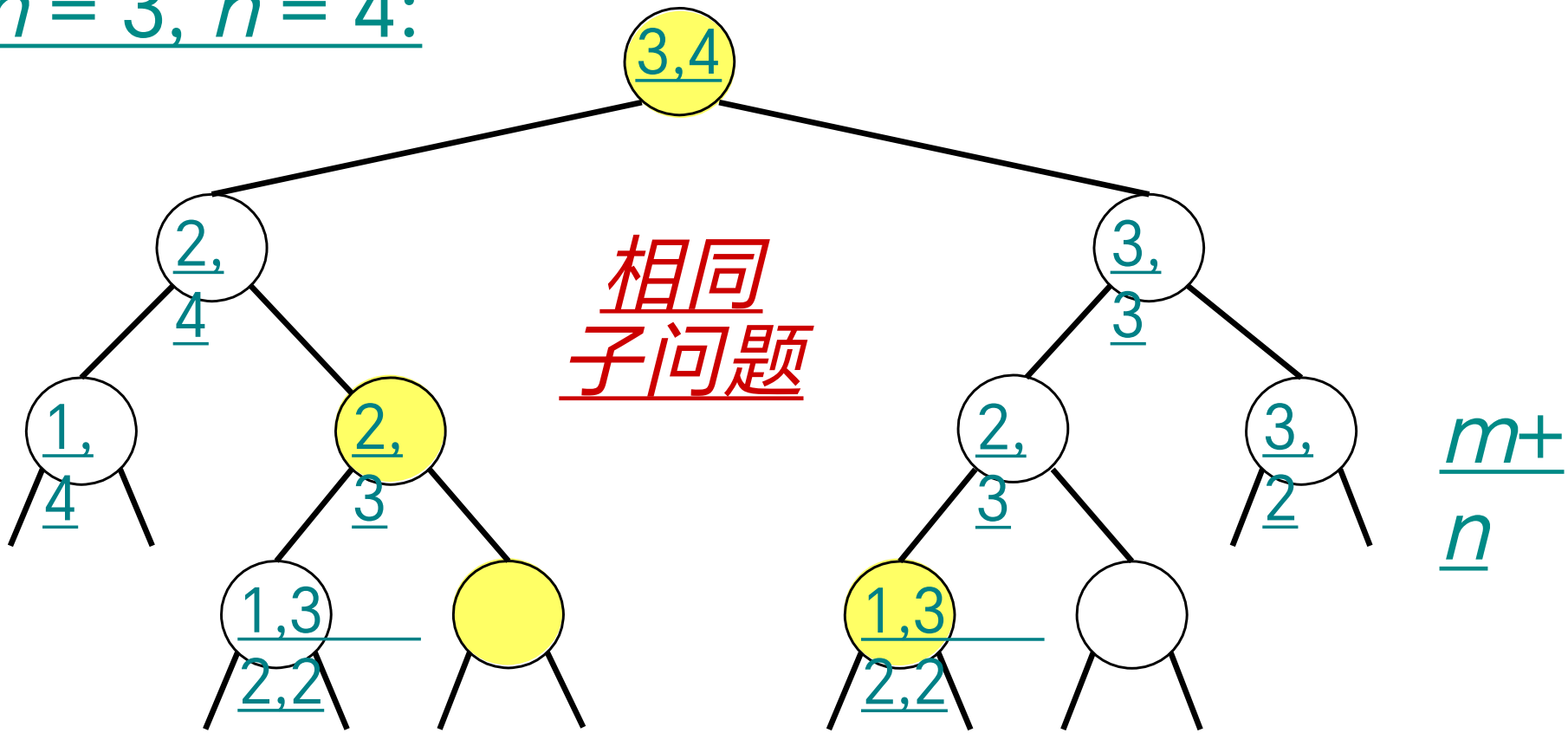
递归树



高度 = $m + n$ 潜在的工作量是指数的.

递归树

$m = 3,\ n = 4$:



高度 $= m + n$ ⇒ 潜在的工作量是指数的。
但是有些子问题我们已经解决了!

动态规划标志#2

重叠子问题

一个递归方案包括重复出现的不同的子问题。

动态规划标志#2

重叠子问题

一个递归方案包括重复出现的不同的子问题。

两个长度为 m 和 n 的字符串的不同的LCS子问题的数目仅仅有 mn个。

LCS的备忘录算法

备忘录: 在计算出一个子问题的答案后，将其存储在一个表中。此后调用将检查表，以避免重复工作。

LCS的备忘录算法

备忘录: 在计算出一个子问题的答案后，将其存储在一个表中。此后调用将检查表，以避免重复工作。

LCS(x, y, i, j)

  if $c[i, j]$ = NIL

    then if $x[i] = y[j]$

      then $c[i, j] \leftarrow$ LCS(x, y, i–1, j–1) + 1

      else $c[i, j] \leftarrow \max\{$LCS($x$, y, i–1, j),

        LCS(x, y, i, j–1)$\}$

和以前一样

LCS的备忘录算法

备忘录: 在计算出一个子问题的答案后，将其存储在一个表中。此后调用将检查表，以避免重复工作。

LCS(x, y, i, j)

  if $c[i, j]$ = NIL

    then if $x[i] = y[j]$

      then $c[i, j] \leftarrow$ LCS(x, y, i–1, j–1) + 1

      else $c[i, j] \leftarrow \max\{$LCS($x$, y, i–1, j),

        LCS(x, y, i, j–

和以前一样

时间 = $\Theta(mn)$ = 每个表项常量工作 。

空间 = $\Theta(mn)$.

LCS的动态规划算法-自底向上

思路:

由底向上计算表.

i	x_i	j 0	1	2	3	4	5	6
		y_j A	B	D	C	A	B	
0		0	0			0	00	00
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS的动态规划算法-自底向上

思路:

由底向上计算表.

i	x_i	j 0	1	2	3	4	5	6
		y_j A	B	D	C	A	B	
0		0	0			0	00	00
1	A	0	0	0	0	1	1	1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS的动态规划算法-自底向上

思路:

由底向上计算表.

i	x_i	j 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0		0	0			0	00	00
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	
7	B	0	1	2	2	3	4	4

LCS的动态规划算法-自底向上

思路:
由底向上计算表.

问题:
如何获取求解方案.

i	x_i	j 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0		0	0			0	00	00
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	
7	B	0	1	2	2	3	4	

LCS的动态规划算法-自底向上

思路:
由底向上计算表.

问题:
如何获取求解方案.

i	x_i	j 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0		0	0		0	00	00	
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	
7	B	0	1	2	2	3	4	
								4

LCS的动态规划算法-自底向上

思路:
由底向上计算表.

问题:
如何获取求解方案.

i	x_i	j = 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0		0	0		0	00	00	
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	
7	B	0	1	2	2	3	4	

LCS的动态规划算法-自底向上

思路:
由底向上计算表.

问题:
如何获取求解方案.

思考:
如何获取所有求解方案.

i		j 0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0		0	00	00	
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	
7	B	0	1	2	2	3	4	

LCS的动态规划算法-自底向上

- 找到解决子问题的正确顺序。
- 为了计算 $c[i, j]$, 我们需要 $c[i-1, j-1]$, $c[i-1, j]$, 和 $c[i, j-1]$.
- $b[i, j]$: 指向表 w.r.t. 计算 $c[i, j]$时选择的最优方案.

```
LCS-Length(X,Y)
m ← length[X];
n ← length[Y];
for i ← 1 to m
    c[i, 0] ← 0;
for j ← 0 to n
    c[0, j] ← 0;
for i ← 1 to m
    for j ← 1 to n
        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] ← " ← "
return c and b
```

LCS举例

- 从 $b[m, n]$ 回溯到 $b[1, 1]$, 跟着箭头需要: $O(m+n)$ 的时间.

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0			0	00	00
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	
7	B	0	1	2	2	3	4	

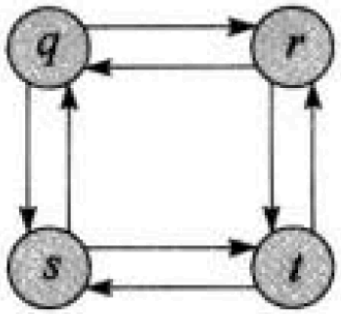
延伸：关于最优子结构的判定

- 给定有向图G=(V, E)和两个顶点*u*，*v*。考虑如下两个问题是否具有最优子结构。

- 无权最短路径：找到一条从*u*到*v*边数最少的路径
- 无权最长路径：找到一条从*u*到*v*的边数最多的简单路径

无权最长路径是否具有最优子结构?

- 不仅不具备最优子结构性质
- 子问题的解的组合甚至都不是原问题的合法解
- 原因?



此例显示了无权有向图最长简单路径问题不具有最优子结构性质。路径 $q \to r \to t$ 是从 q 到 t 的一条最长简单路径，但 $q \to r$ 不是从 q 到 r 的一条最长简单路径，$r \to t$ 同样不是从 r 到 t 的一条最长简单路径

- 子问题无关：
- 　　同一个原问题的一个子问题的解不影响另一个子问题的解。

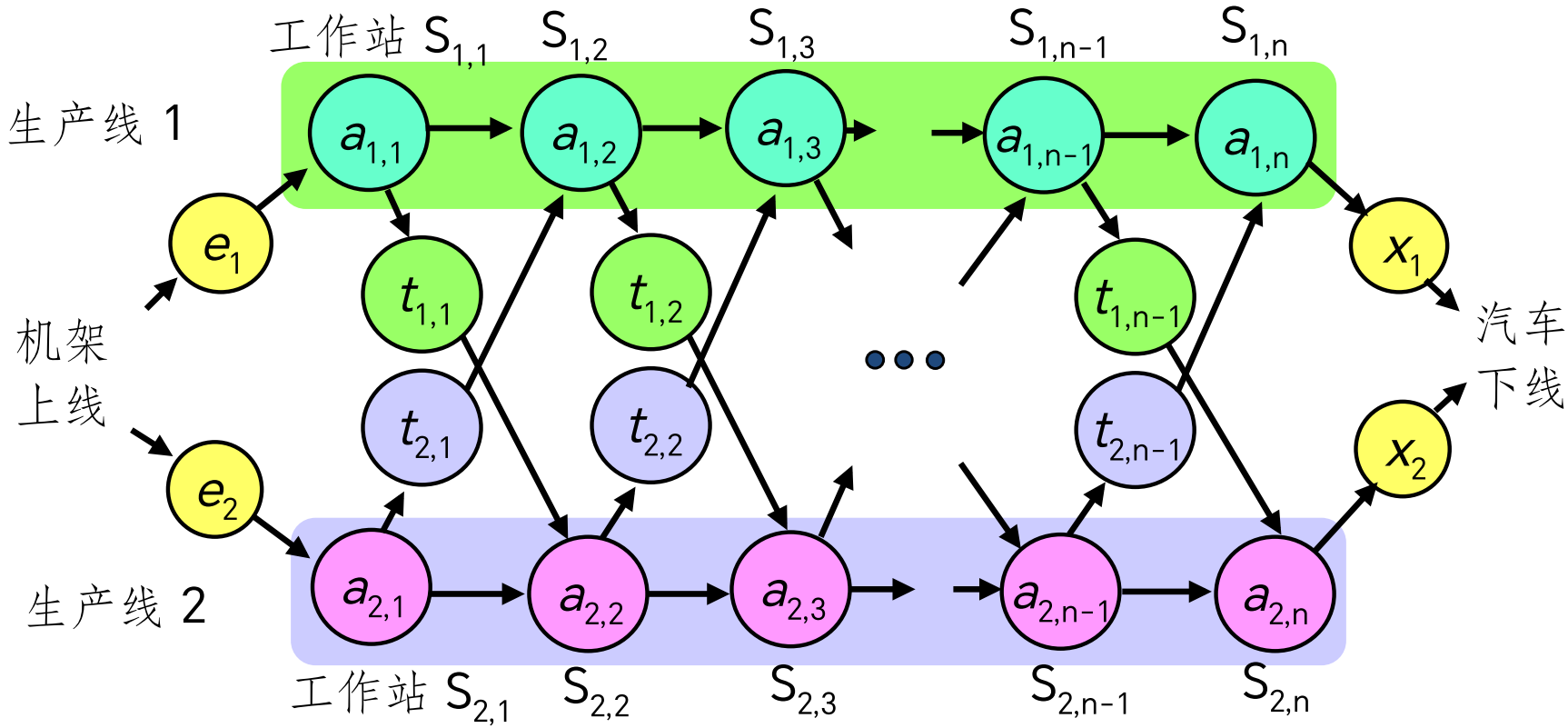
- 最短路径的子问题：相互不共享资源

延伸：最长上升子序列问题

- 最长上升子序列（longest increasing subsequence），也可以叫最长非降序子序列，简称LIS

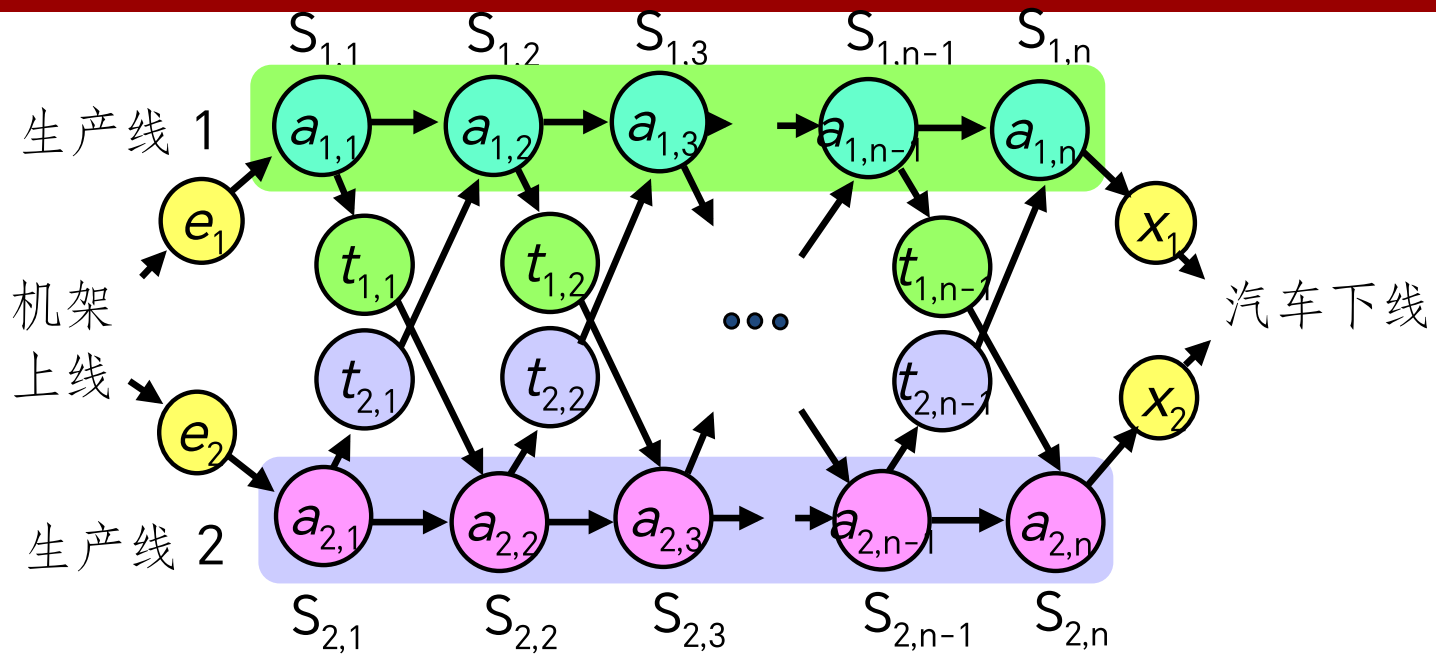
- 例：6个数分别是：１７６２３４，求最长上升子序列

习题4：装配线调度



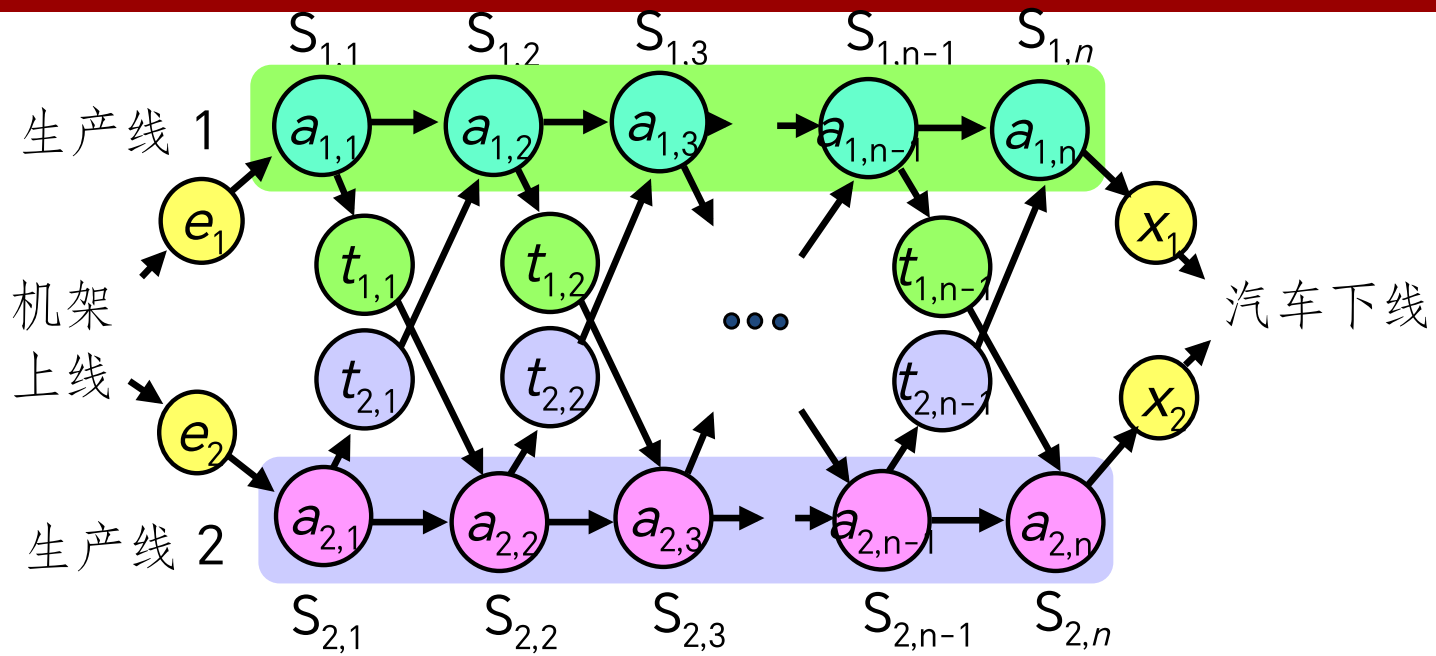
- 一个汽车机架计入一个生产线，在每个工作站增加一些配件，最后在装配线结束，完成的汽车下线.
  - $S_{i,j}$: 生产线 i上的第j个工作站
  - $a_{i,j}$:工作站$S_{i,j}$需要的装配时间
  - $t_{i,j}$: 每条线上从工作站 $S_{i,j}$ 到另外一条线上的 j+1 个站的传输时间.
  - e_i (x_i):生产线i上从上线到下线的时间

最优子结构



- 目标：确定选择哪些工作站使得制造一台汽车总的时间最小化。
  - 蛮力法: $\Omega(2^n)$, 为什么？
  - 问题是线性有序的 => 动态规划?
- **最优子结构:**假设经$S_{1,j}$最快的线路通过$S_{1,j-1}$，那么装配机架必定利用了从开始点到$S_{1,j-1}$的最快路径.

重叠子问题: 递归



- **重叠子问题：** 通过工作站$S_{1,j}$最快的路线要么是通过$S_{1,j-1}$然后是$S_{1,j}$，或者通过 $S_{2,j-1}$ 然后传输到线 1 到达 $S_{1,j}$.
- $f_i[j]$: 从开始点通过 $S_{i,j}$最快的时间

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

- 通过整个工厂的最快时间为 $f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$

计算最快时间

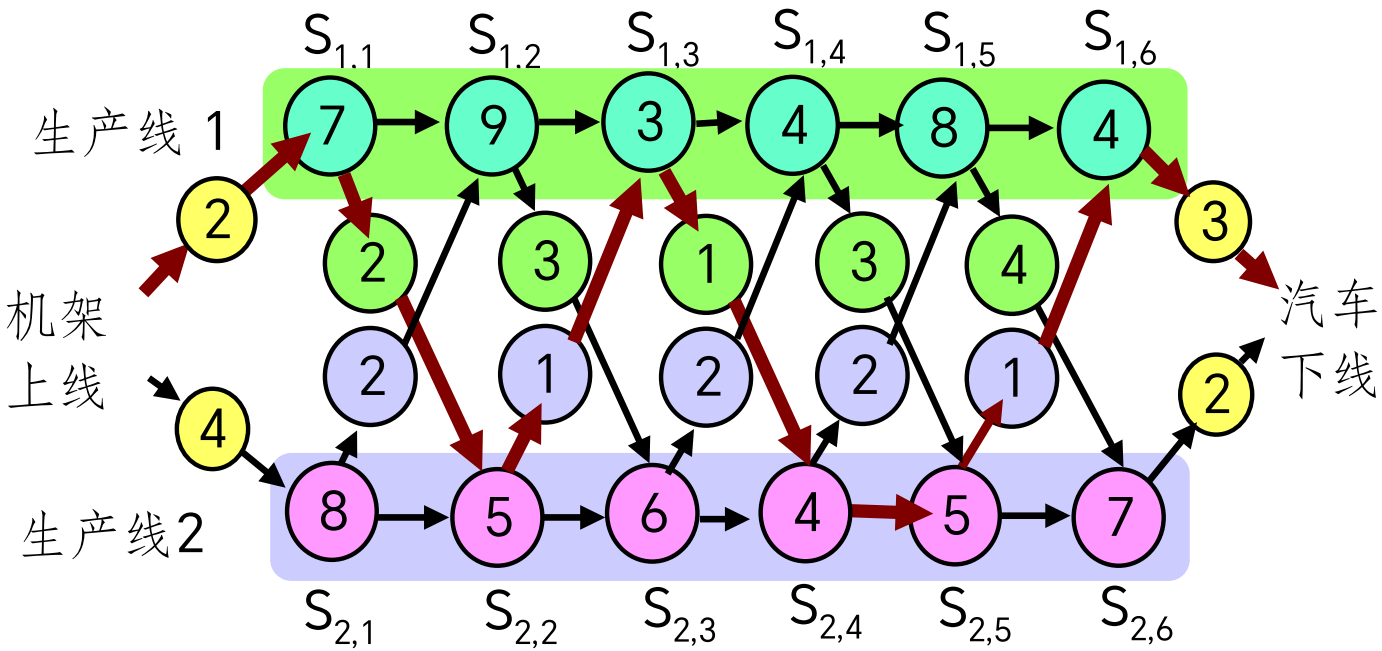
```
Fastest-Way(a, t, e, x, n)
f1[1] ← e1 + a1,1
f2[1] ← e2 + a2,1
for j ← 2 to n
    do if f1[j−1] + a1,j ≤ f2[j−1] + t2,j −1 + a1,j
            then f1[j] ← f1[j−1] + a1,j
                 l1[j] ← 1
            else f1[j] ← f2[j−1] + t2,j −1 + a1,j
                 l1[j] ← 2
    do if f2[j−1] + a2,j ≤ f1[j−1] + t1,j −1 + a2,j
            then f2[j] ← f2[j−1] + a2,j
                 l2[j] ← 2
            else f2[j] ← f1[j−1] + t1,j −1 + a2,j
                 l2[j] ← 1
if f1[n] + x1 ≤ f2[n] + x2
  then f* = f1[n] + x1
       l* = 1
  then f* = f2[n] + x2
       l* = 2
```

- $l_i[j]$: 为以最快路线通过 $S_{i,j}$时第 j−1个站点线号

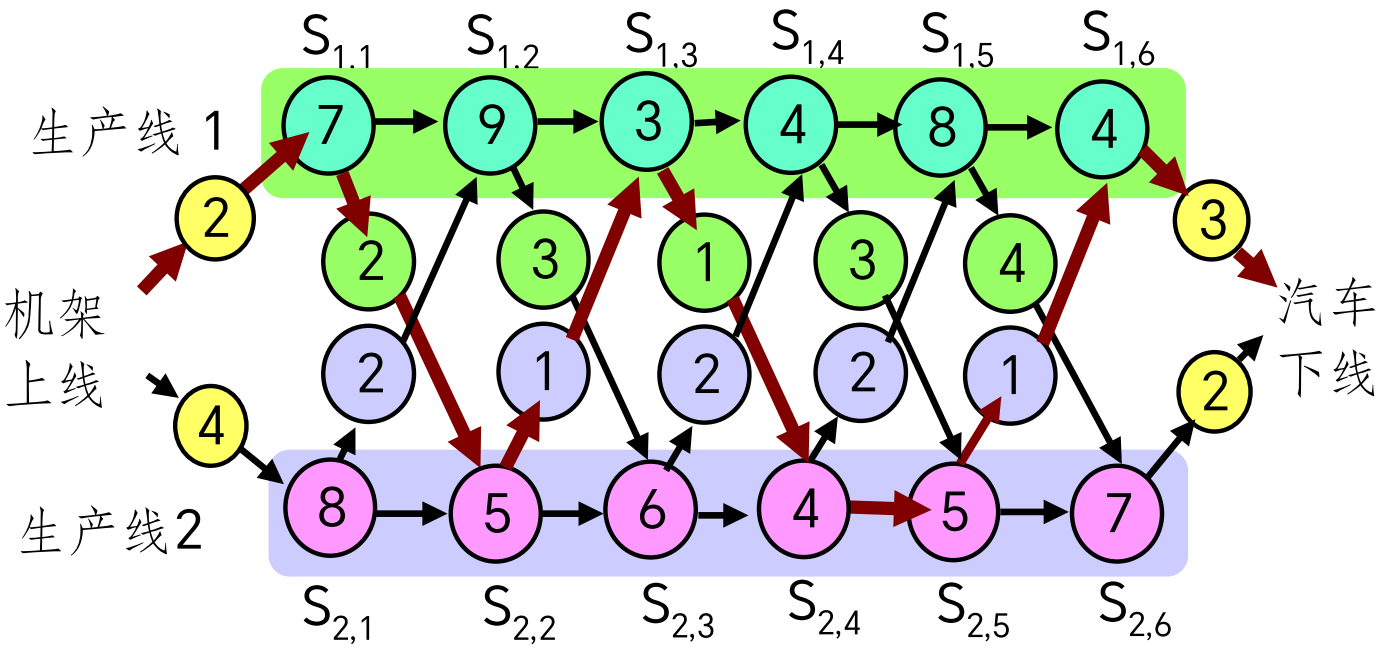
创建最快路径

```
Print-Station(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
```

生产线 1, 工作站 6
生产线 2, 工作站 5
生产线 2, 工作站 4
生产线 1, 工作站 3
生产线 2, 工作站 2
生产线 1, 工作站 1



装配线调度示例



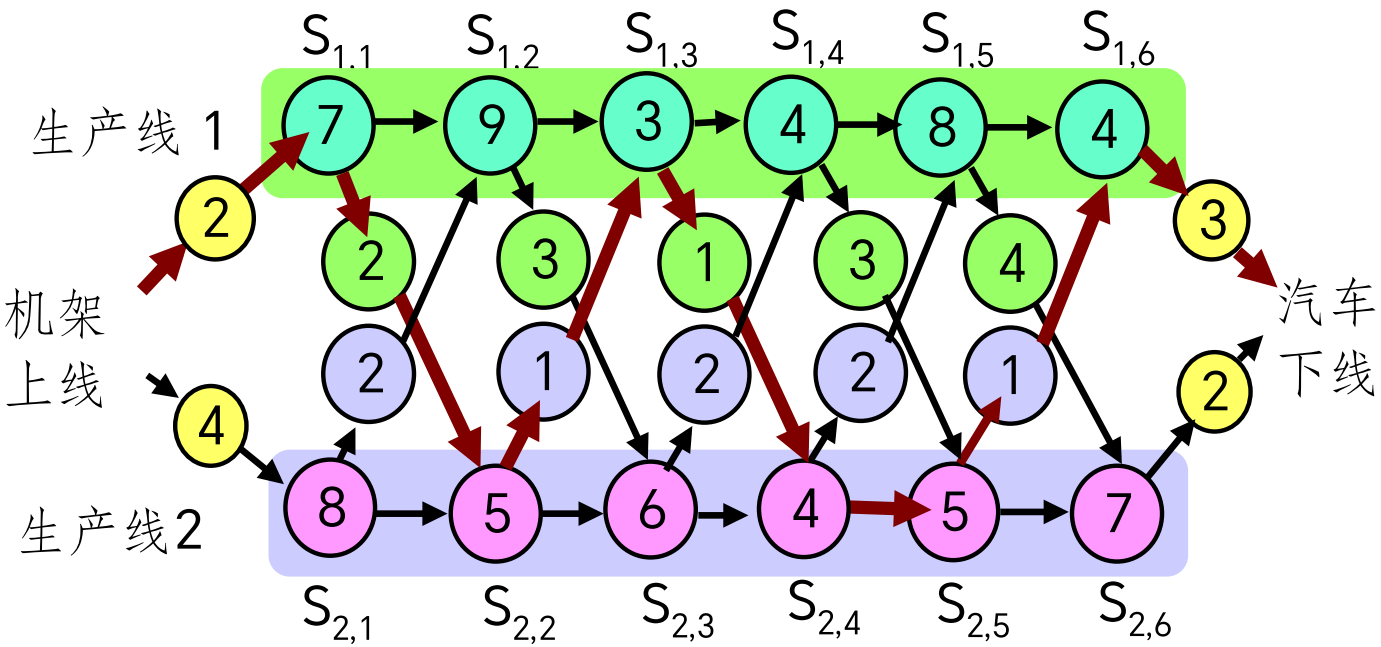
$f_1[j]$						
$f_2[j]$						

$f^* = ?$

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

$l^* = ?$

装配线调度示例



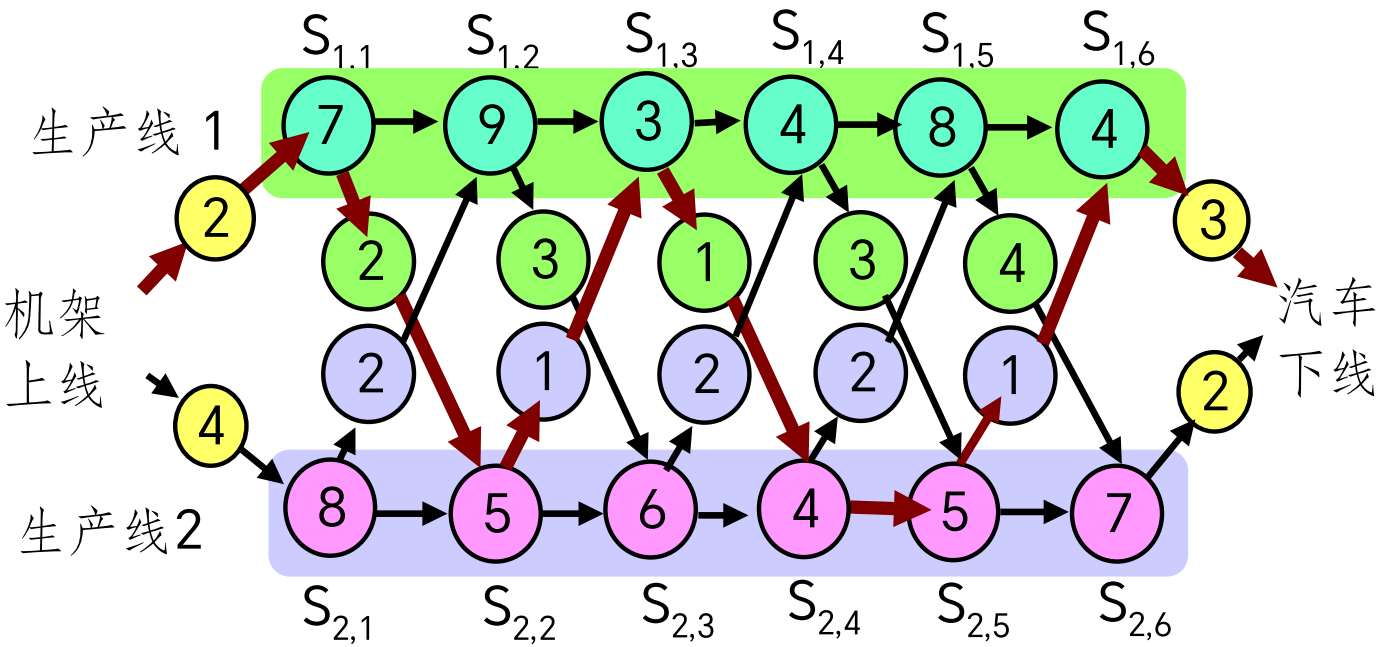
$f_1[j]$	9					
$f_2[j]$	12					

$f^* =$

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

$l^* =$

装配线调度示例



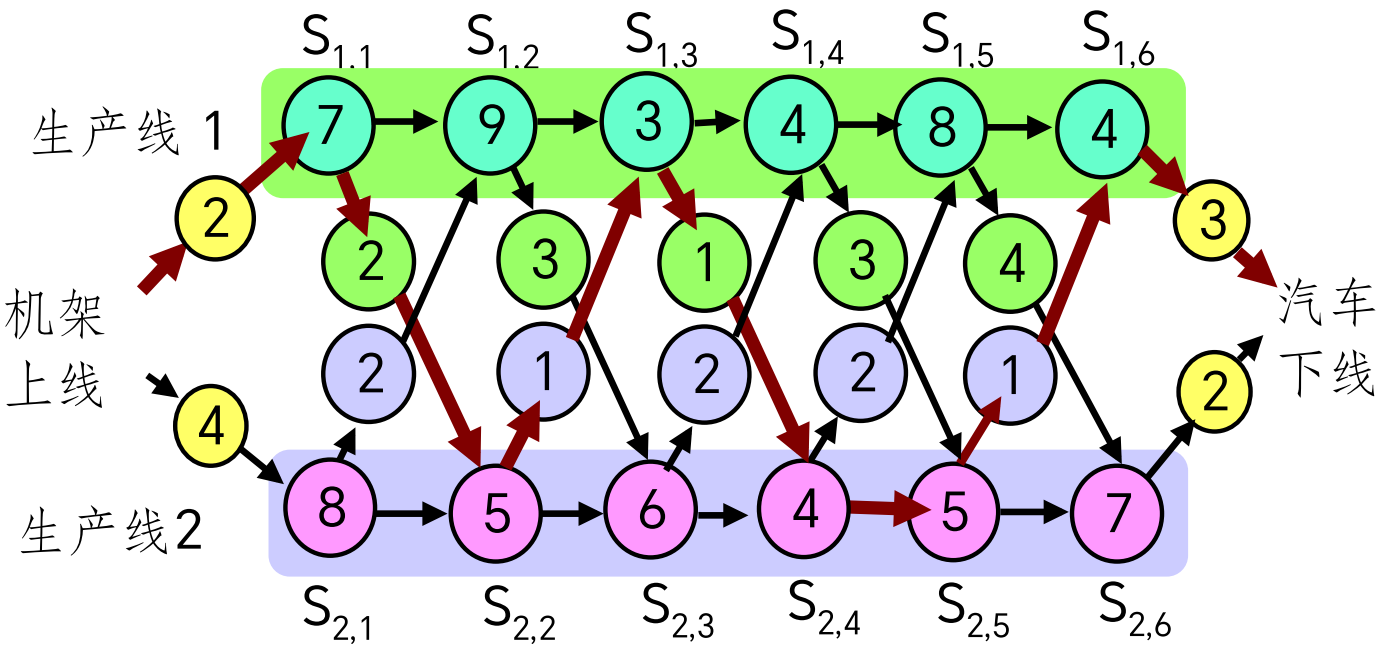
$f_1[j]$	9	18				
$f_2[j]$	12	16				

$f^* =$

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

$l^* =$

装配线调度示例



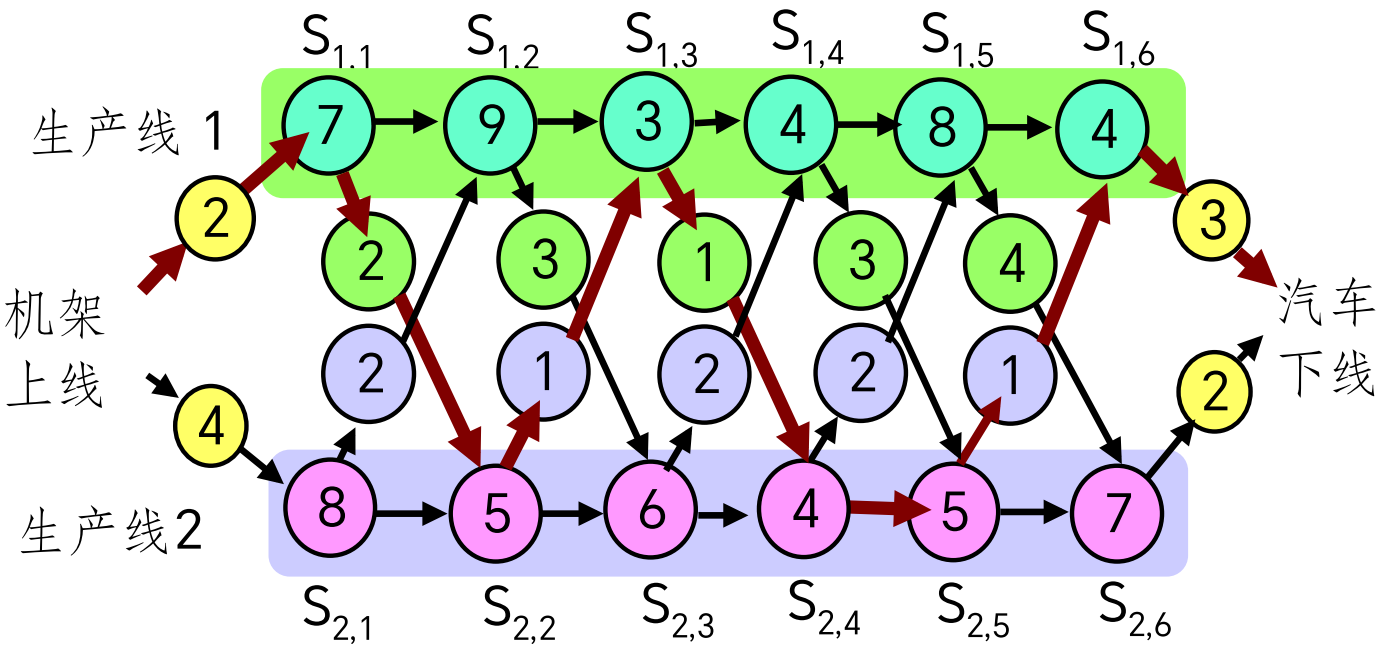
$f_1[j]$	9	18	20			
$f_2[j]$	12	16	22			

$f^* =$

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

$l^* =$

装配线调度示例



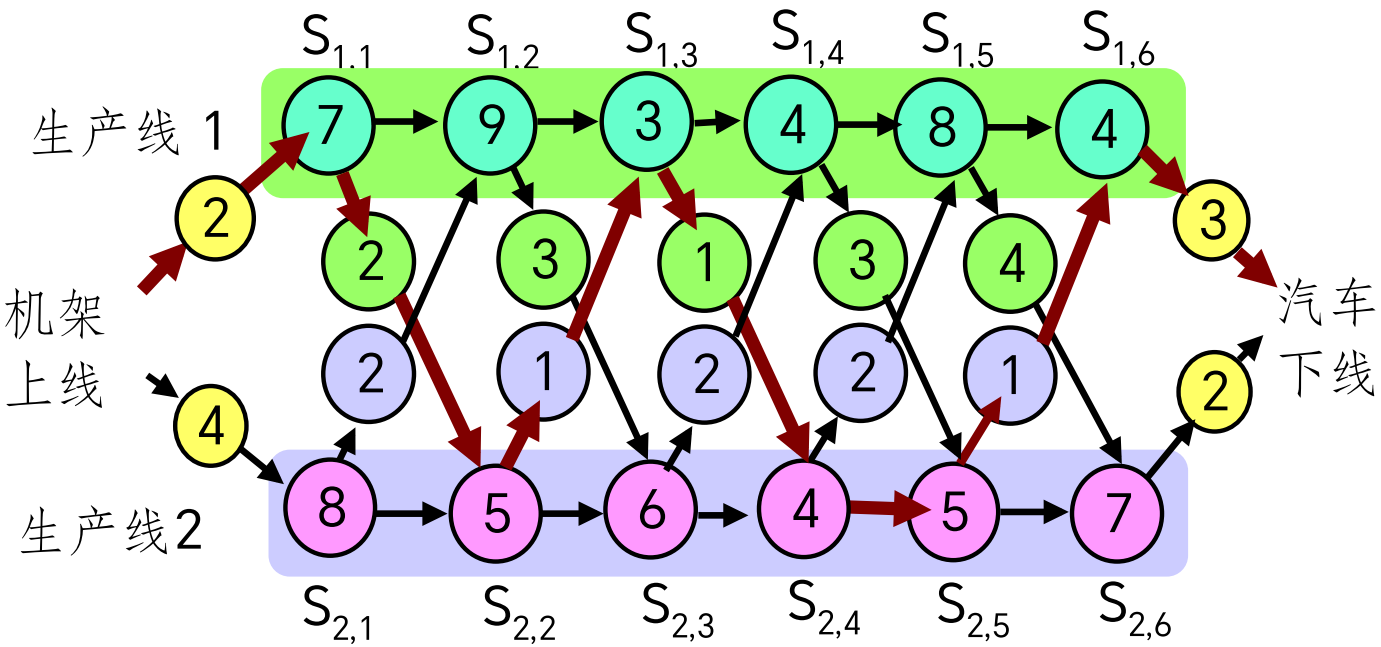
$f_1[j]$	9	18	20	24		
$f_2[j]$	12	16	22	25		

$f^* =$

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

$l^* =$

装配线调度示例



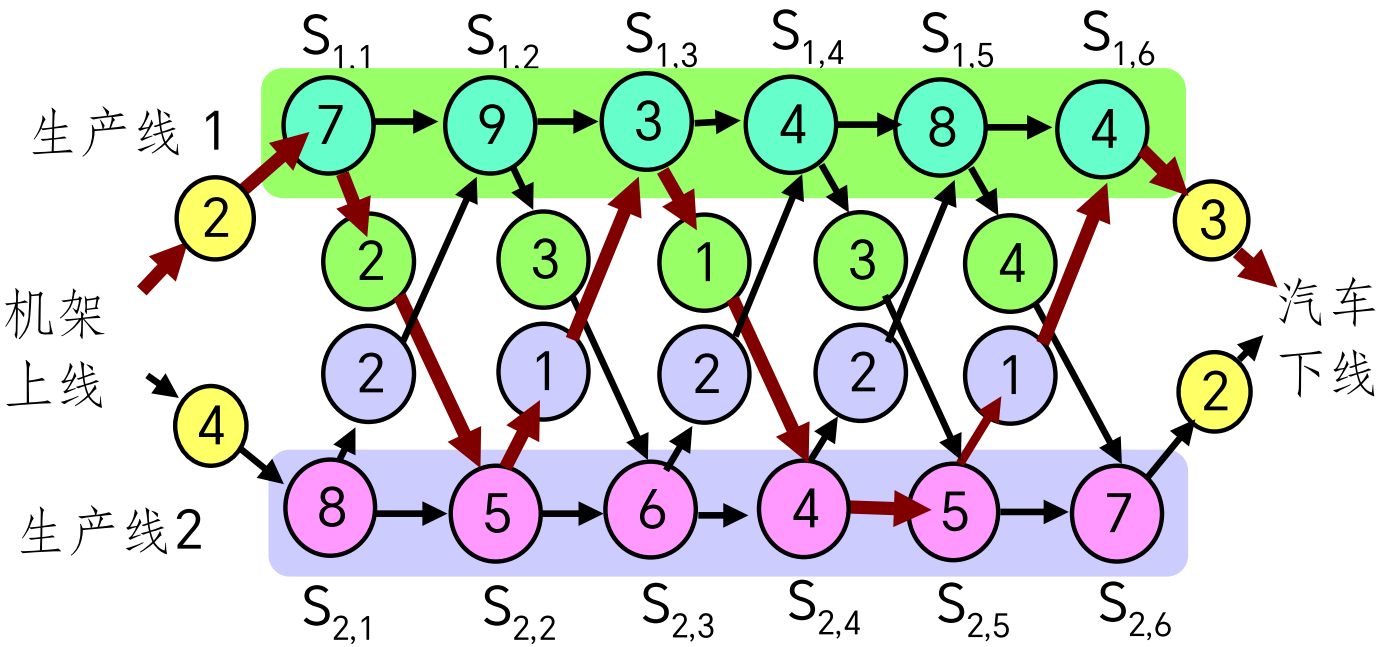
$f_1[j]$	9	18	20	24	32	
$f_2[j]$	12	16	22	25	30	

$f^* =$

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

$l^* = 1$

装配线调度示例



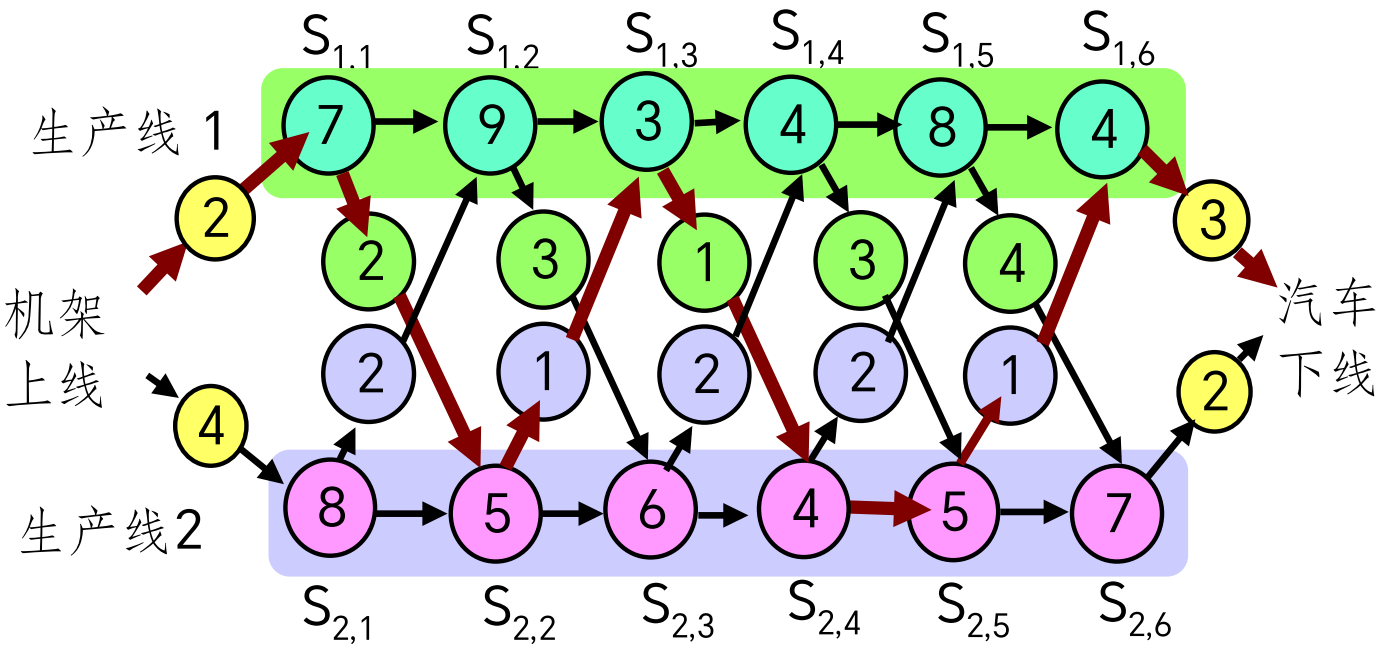
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* =$

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

$l^* =$

装配线调度示例



$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

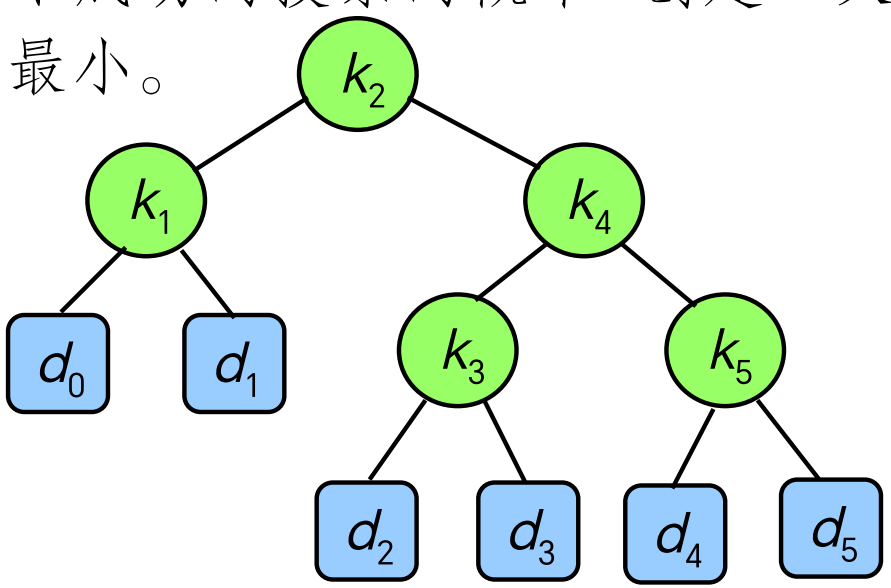
$f^* = 38$

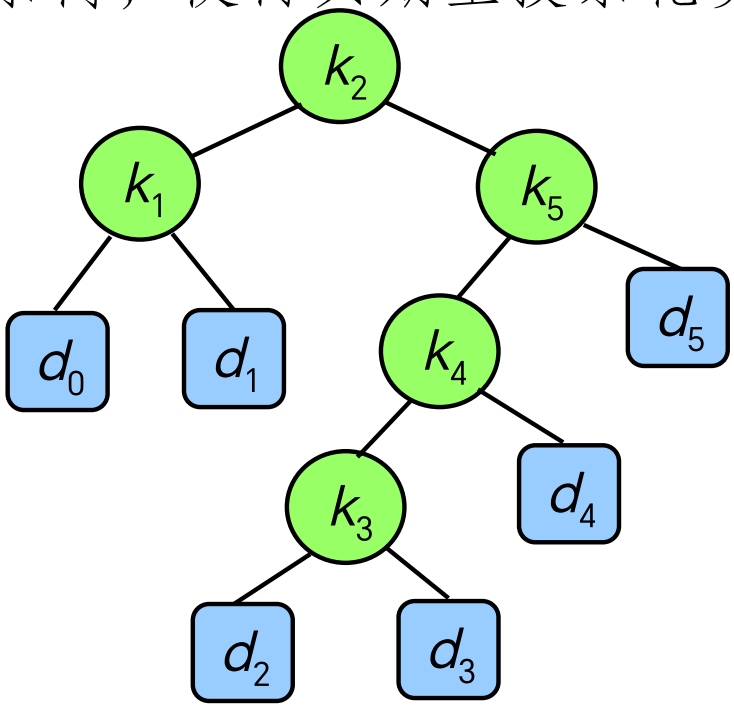
j	1	2	3	4	5	6
$l_1[j]$	1	2	1	1	2	
$l_2[j]$	1	2	1	2	2	

$l^* = 1$

习题5：最优二叉搜索树

- 给定一个序列 $K=<k_1, k_2, \ldots, k_n>$ 有n个有序且各不相同的键 $(k_1 < k_2 < \ldots < k_n)$， 集合 $P=<p_1, p_2, \ldots, p_n>$表示在$K$中成功的搜索的概率; $D=<d_0, d_1, d_2, \ldots, d_n>$ 为$n+1$ 个不同的哑键，d_i 表示所有在 k_i和k_{i+1}之间的值,$Q=<q_0, q_1, q_2, \ldots, q_n>$ 表示不成功的搜索的概率. 创建二叉搜索树，使得其期望搜索花费最小。

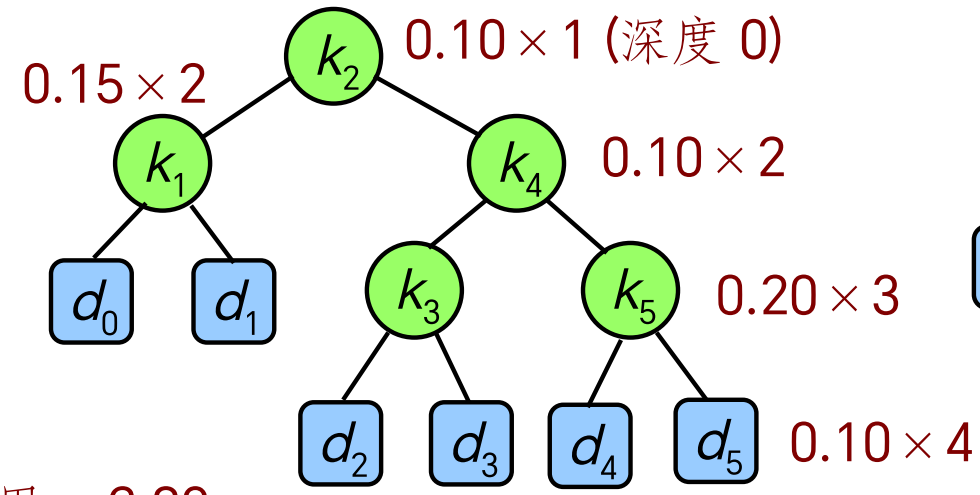




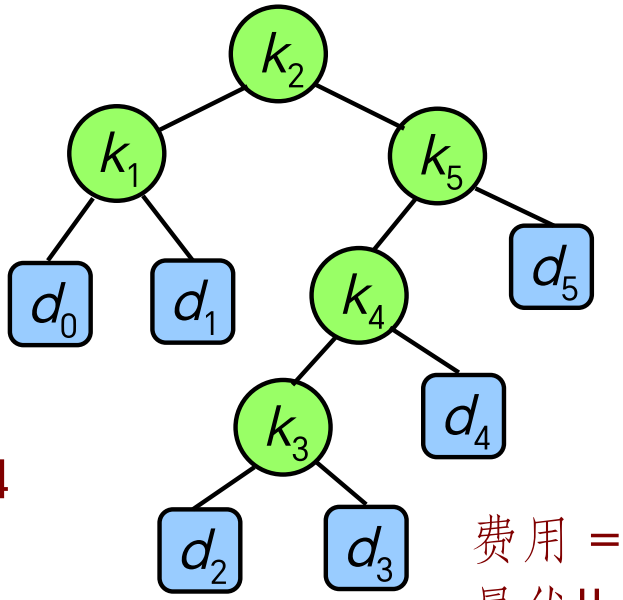
一个例子

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

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



费用 = 2.80



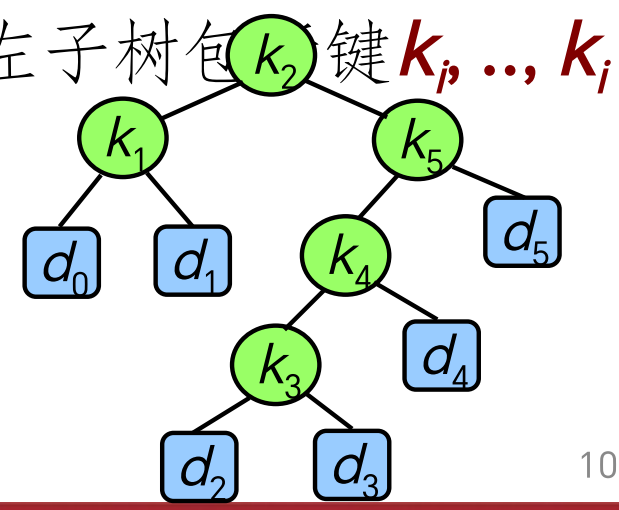
费用 = 2.75
最优!!

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

$$= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i$$

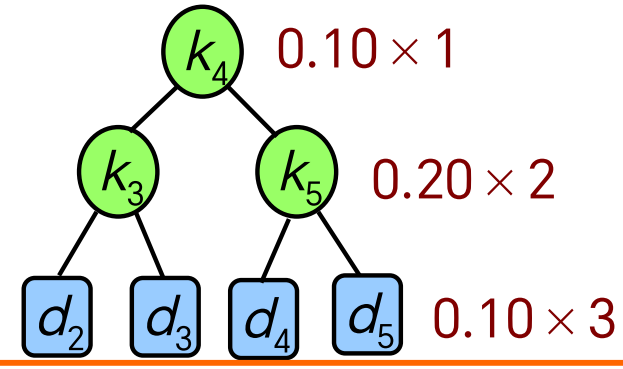
最优子结构

- 如果一棵最优二叉搜索树T的子树T'含有键 $k_i, \ldots, k_j$, 那么这个子树T'肯定是子问题键 $k_i, \ldots, k_j$和哑键$d_{i-1}, \ldots, d_j$的最优解。(请自行证明)
- 解的结构
  - 给定键$k_i, \ldots, k_j$以 k_r $(i \le r \le j)$ 为根, 其左子树包括键 $k_i, \ldots, k_{r-1}$(和哑键$d_{i-1}, \ldots, d_{r-1}$)同时其右子树包括键 $k_{r+1}, \ldots, k_j$(和哑键 $d_r, \ldots, d_j$).
  - 对于包括键$k_i, \ldots, k_j$和根k_i的子树, 左子树包括键$k_i, \ldots, k_{i-1}$ (没有键) 和哑键 d_{i-1}.



重叠子问题: 递归

- $e[i, j]$: 搜索一棵包括键$k_i, \ldots, k_j$的最优二叉搜索树的期望费用.
  - 希望找出 $e[1, n]$
  - $e[i, i-1] = q_{i-1}$ (仅对哑键 d_{i-1}).



$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

- 如果$k_r (i \le r \le j)$是包括键$k_i, \ldots, k_j$的最优子树的根，并且令

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$
$$= e[i, r-1] + e[r+1, j] + w(i, j)$$

- 递归:

$$e[i, j] = \begin{cases} q_{i-1} & if \ \ j = i-1 \\ \min\limits_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w(i, j) \} & if \ \ i \le j \end{cases}$$

计算最优费用

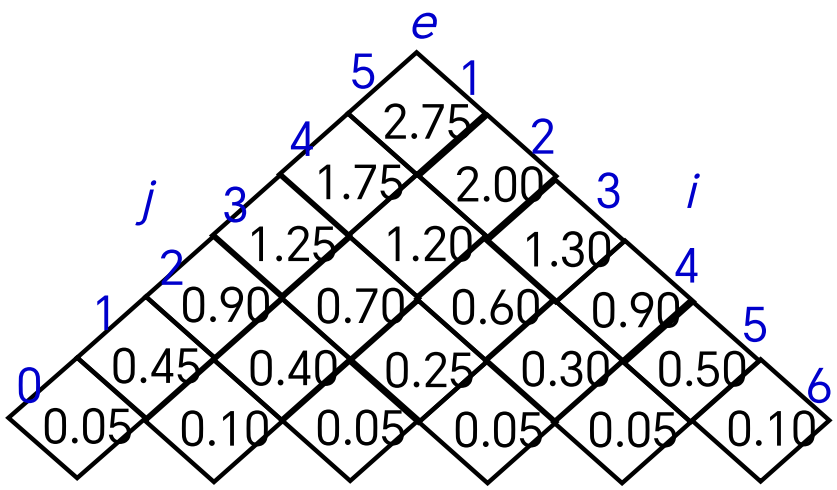
- $e[i, j]$需要一个表 $e[1..n+1, 0..r$
- 迭代计算 $w(i, j)$ (为什么?)

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

$$w[i, j] = \begin{cases} q_{i-1} & if\ \ j = i-1 \\ w[i, j-1] + p_j + q_j & if\ \ i \le j \end{cases}$$

```
Optimal-BST(p, q, n)
for i ← 1 to n + 1
    e[i, i-1] ← q_{i-1}
    w[i, i-1] ← q_{i-1}
for l ← 1 to n
    for i ← 1 to n - l + 1
        j ← i + l - 1
        e[i, j] ← ∞
        w[i, j] ← w[i, j-1] + p_j + q_j
    for r ← i to j
        t ← e[i, r-1] + e[r+1, j] + w[i, j]
        if t < e[i, j]
            e[i, j] ← t
            root[i, j] ← r
return e and root
```

.$root[i, j]$: 索引r对应
k_r 为最优搜索树的根，
该树包括 键 $k_i, \ldots, k_j$.

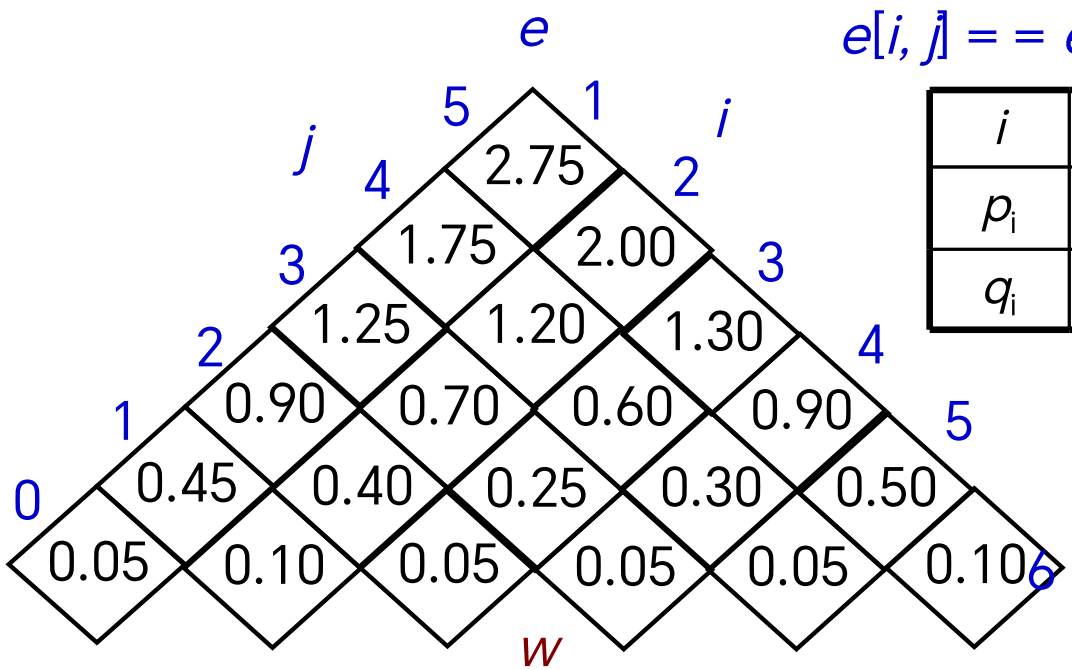


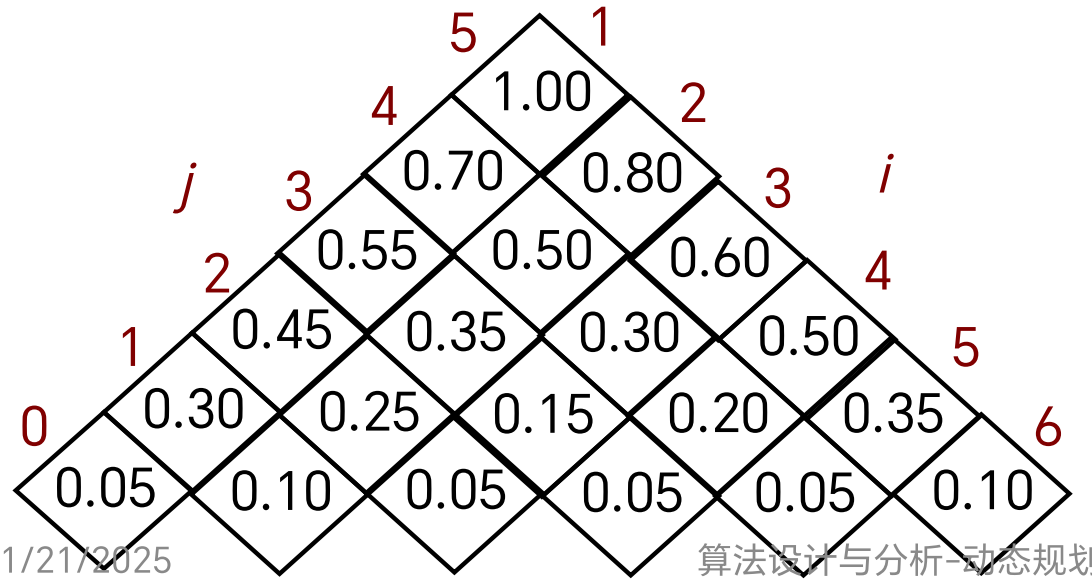
举例

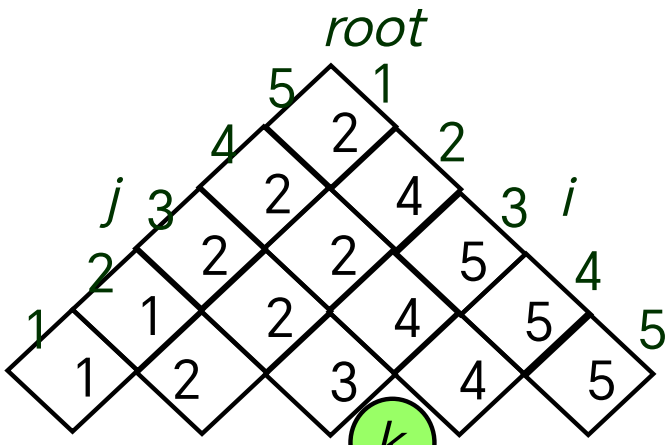
$$w[i,j] = \begin{cases} q_{i-1} & if\ \ j = i-1 \\ w[i,j-1] + p_j + q_j & if\ \ i \le j \end{cases}$$

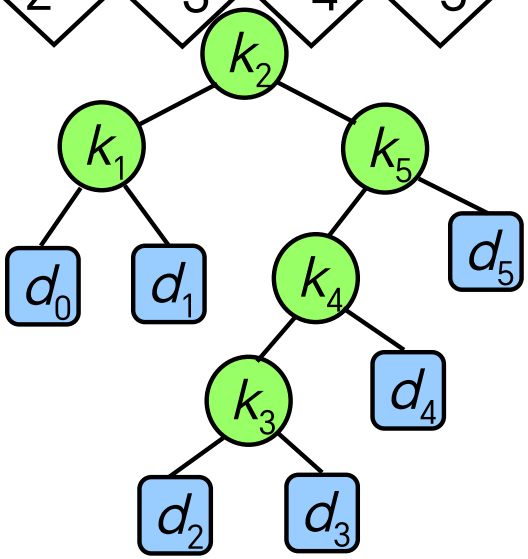
$e[i, j] = = e[i, r-1] + e[r+1, j] + w(i, j)$

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10









合肥工業大學
HEFEI UNIVERSITY OF TECHNOLOGY

Q/A?

DMIC
-HFUT-
合肥工业大学数据挖掘与智能计算“千人计划”研究团队
HFUT Data Mining and Intelligent Computing Laboratory