
动态规划

Dynamic Programming

刘 铎

liuduo@bjtu.edu.cn

斐波那契数列

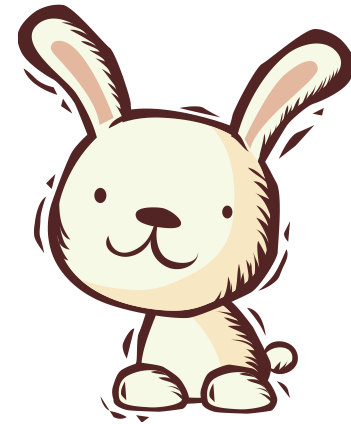
Fibonacci Number

示例：斐波那契数列



(image of Leonardo Fibonacci from
<http://www.math.ethz.ch/fibonacci>)

- $f_1 = 1, f_2 = 1$
- 对于整数 $n > 2$:
$$f_n = f_{n-1} + f_{n-2}$$
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- $f_n = ?$



示例：斐波那契数列

- $f_{200} = ?$
- $f_n = ?$ 其中 n 是任一正整数

效率极低

Algorithm 1 $F(n)$

if $n \leq 2$ **then**

return 1

else

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

示例：斐波那契数列

■ f(18)=2584	time:0.00000s	■ f(39)=63245986	time:0.41900s
■ f(19)=4181	time:0.00100s	■ f(40)=102334155	time:0.69000s
■ f(20)=6765	time:0.00000s	■ f(41)=165580141	time:1.06600s
■ f(21)=10946	time:0.00000s	■ f(42)=267914296	time:1.81500s
■ f(22)=17711	time:0.00000s	■ f(43)=433494437	time:2.82000s
■ f(23)=28657	time:0.00000s	■ f(44)=701408733	time:4.71800s
■ f(24)=46368	time:0.00100s	■ f(45)=1134903170	time:7.82600s
■ f(25)=75025	time:0.00300s	■ f(46)=1836311903	time:13.26400s
■ f(26)=121393	time:0.01000s	■ f(47)=2971215073	time:22.12000s
■ f(27)=196418	time:0.00300s	■ f(48)=4807526976	time:37.30100s
■ f(28)=317811	time:0.00400s	■ f(49)=7778742049	time:62.18200s
■ f(29)=514229	time:0.00400s	■ f(50)=12586269025	time:90.19200s
■ f(30)=832040	time:0.00500s	■ f(51)=20365011074	time:145.87700s
■ f(31)=1346269	time:0.01300s	■ f(52)=32951280099	time:234.55000s
■ f(32)=2178309	time:0.01900s	■ f(53)=53316291173	time:400.45900s
■ f(33)=3524578	time:0.04000s	■ f(54)=86267571272	time:658.49200s
■ f(34)=5702887	time:0.05100s	■ f(55)=139583862445	time:1039.41400s
■ f(35)=9227465	time:0.06300s		(17.3min)
■ f(36)=14930352	time:0.10100s		
■ f(37)=24157817	time:0.15100s	■ f(200)	time:2.1×10 ³³ s (6.6×10 ²⁵ years)
■ f(38)=39088169	time:0.24200s		

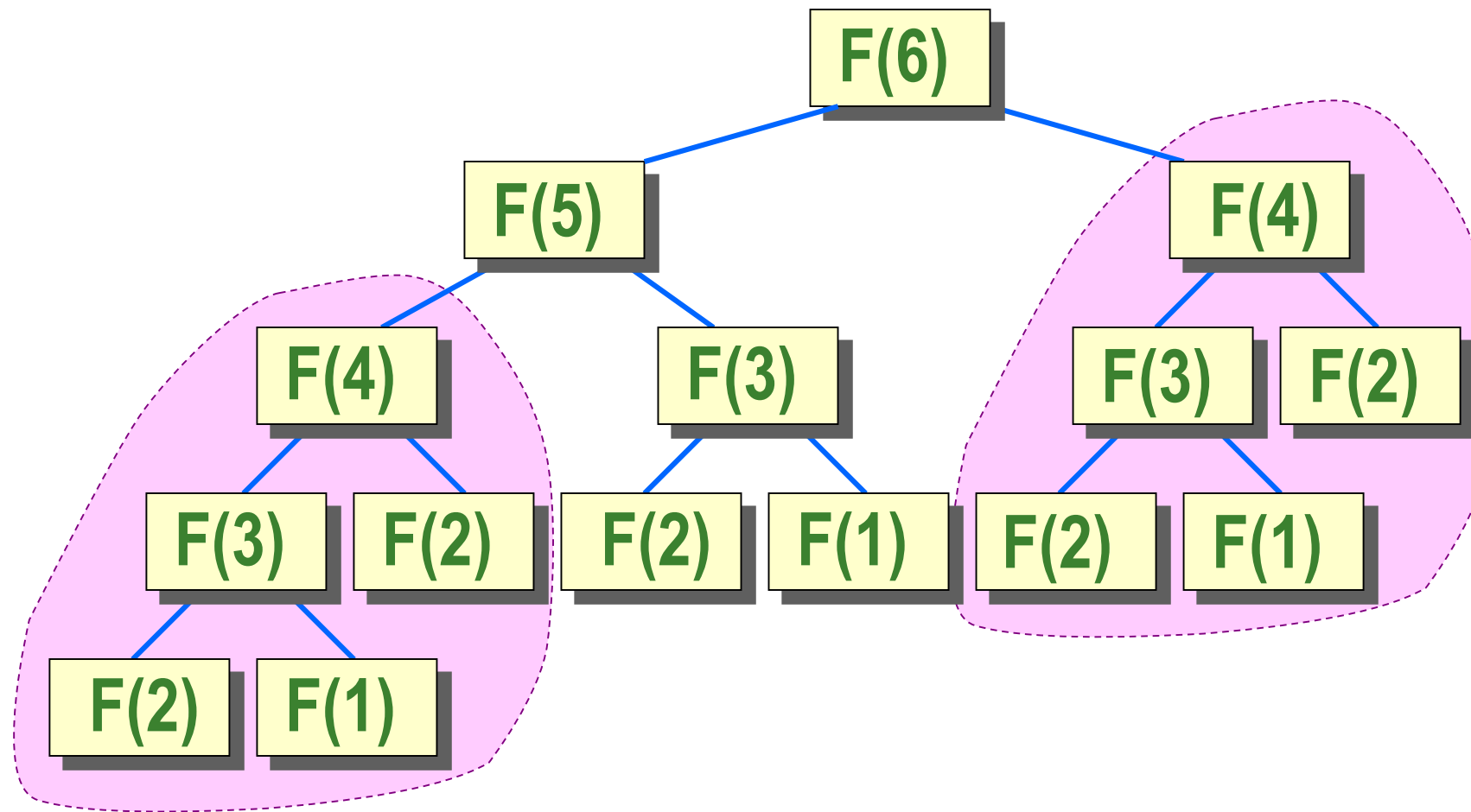
示例：斐波那契数列

- $f_{200} = ?$
- $f_n = ?$ 其中 n 是任一正整数

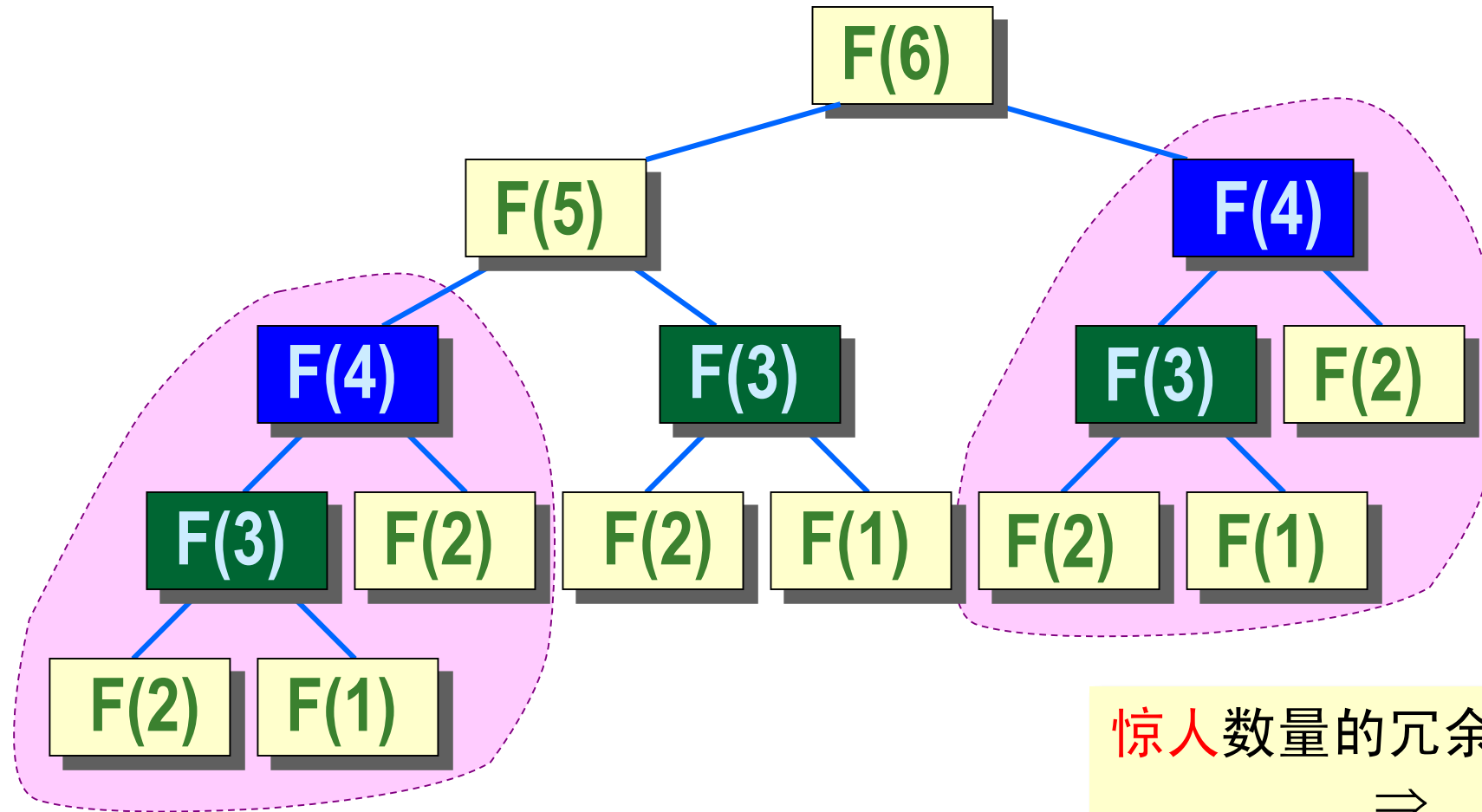
```
Algorithm 1  F ( n )  
if  $n \leq 2$  then  
    return 1  
else  
    return (F( $n-1$ ) + F( $n-2$ )))
```

- 时间复杂度：
 $T(n) = T(n-1) + T(n-2)$
- 为 $O(2^n)$
 - 事实上是 $O(\phi^n)$
 - $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$
- 效率极低
 - 原因在于大量重复计算

示例：斐波那契数列



重叠的子问题

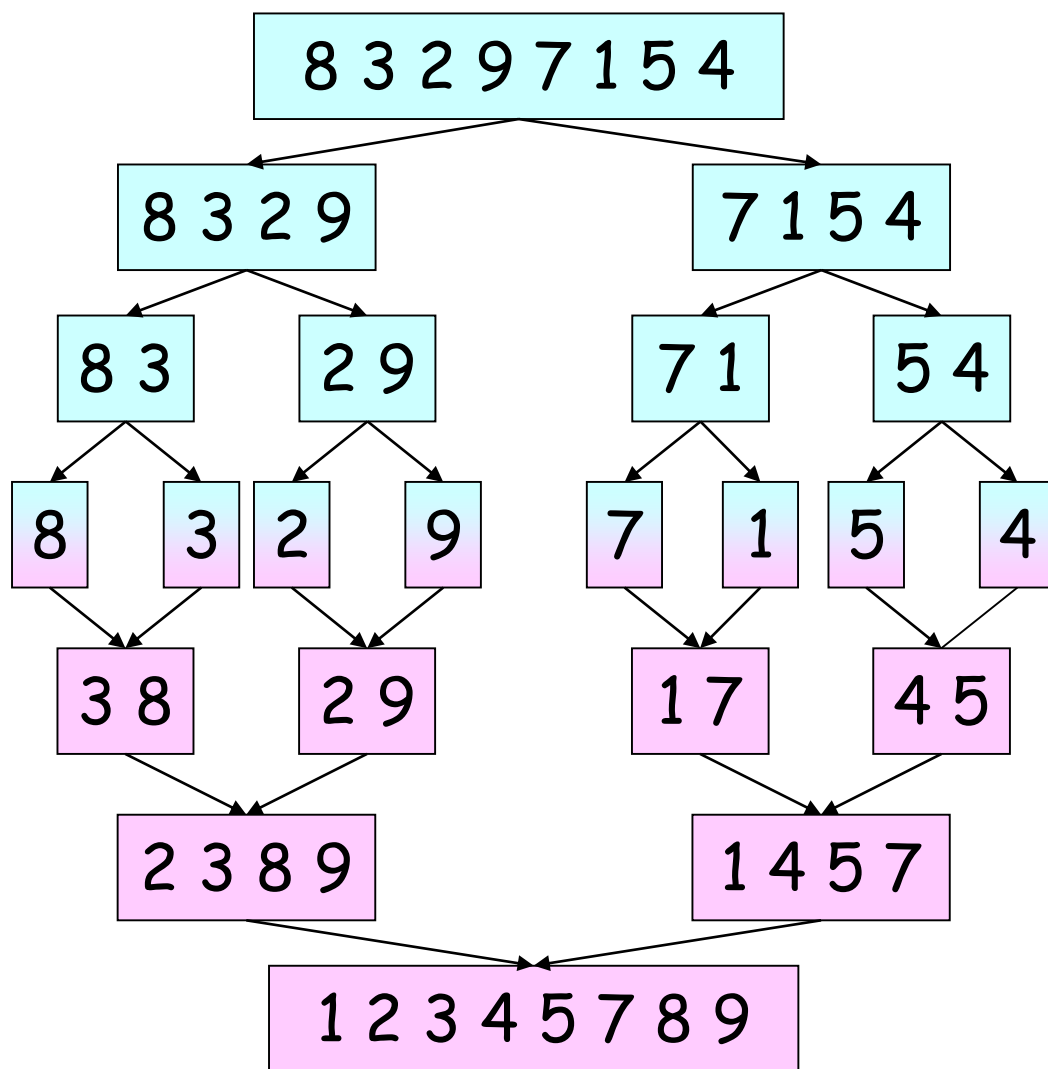


惊人数量的冗余子问题

⇒

指数时间算法

分治策略的子问题？



独立的子问题

⇒

有效的算法

示例：斐波那契数列

- $f_{200} = ?$
- $f_n = ?$ 其中 n 是任一正整数。
- 自下而上计算
- 将值存储以供后续使用
- 这减少了重复计算

Algorithm 2 $F(n)$

注：初始化时创建数组 $F[1:n]$

$F[1] \leftarrow 1, F[2] \leftarrow 1$

for $i = 3$ **to** n **do**

$F[i] \leftarrow F[i-1] + F[i-2]$

return ($F[n]$)

找零问题

Coin Changing



找零问题

- 目标：给定面值分别为1元、5元、10元的硬币（每种都有足够多枚），请设计一个算法，可以使用最少数量的硬币向客户支付给定金额。
 - 此时，贪婪策略（即收银员算法）是有效的
- 如果还有面值为7元的硬币，那么贪婪算法可能就无法得到最优解了
 - 例如凑出19元
 - 贪婪算法需要使用4枚硬币（ $10+7+1+1$ ）
 - 但事实上只需要3枚硬币（ $7+7+5$ ）
- 对于给定的硬币组，如何找到凑成给定金额的最小硬币数？

找零问题

■ 假定**总要**找给顾客第一枚硬币的

1, 5, 7, 10 情况

□ 如果做不到那么也可以直接告知顾客

■ 于是，可以分为以下四种情形：

□ 先给顾客一枚面值1元的硬币，之后再以**最优方式**找给顾客18元

□ 先给顾客一枚面值5元的硬币，之后再以**最优方式**找给顾客14元

□ 先给顾客一枚面值7元的硬币，之后再以**最优方式**找给顾客12元

□ 先给顾客一枚面值10元的硬币，之后再以**最优方式**找给顾客9元

■ 分别计算后，从上述4种方案中选择**最佳方案**

找零问题 —— 递归算法

Find (n)

1. **if** $n = 0$ **then return** 0
2. **if** $n < 0$ **then return** ∞
3. **return min** {
 Find($n - 1$) + 1,
 Find($n - 5$) + 1,
 Find($n - 7$) + 1,
 Find($n - 10$) + 1
}

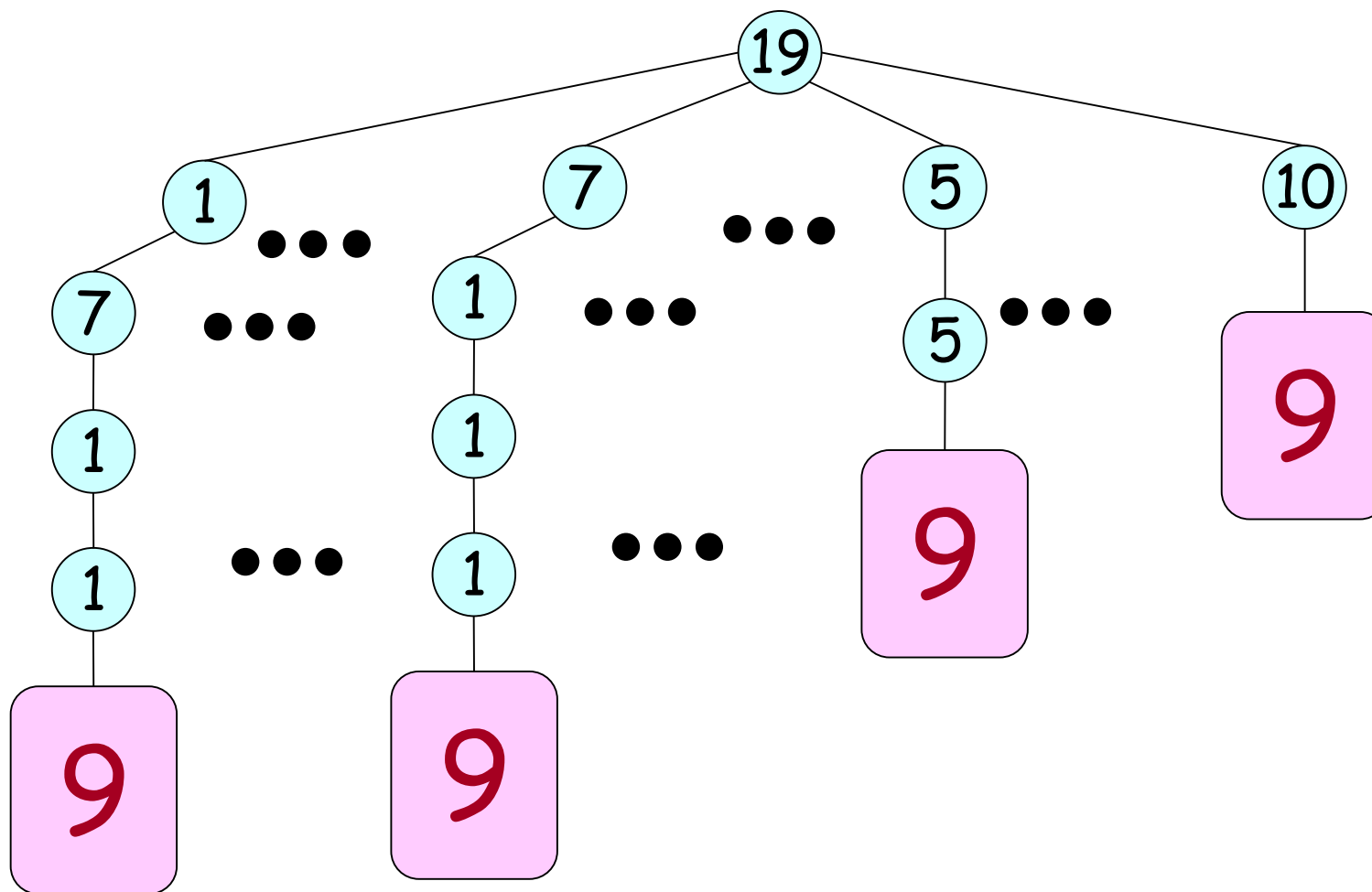
1, 5, 7, 10 情况

$n = 6$ 时?

- 该算法的开销非常大

找零问题 —— 递归算法

1, 5, 7, 10 情况



找零问题

- 目标：给定数值 n 和 m 种硬币的面值 d_1, d_2, \dots, d_m ，设计一种方法，或者用最少数量的硬币凑出总和 n ，或者表明不存在解

- 假设硬币数量是无限的

- 解：

- 令 $d(i)$ 表示凑出总和 i 所需的最少硬币数量
- 初始值： $d(0) = 0$, $d(i) = \infty$ ($i \neq 0$)
- 递推式： $d(i) = \min \{ d(i - d_k) \} + 1$

$$k = 1, 2, \dots, m$$

$$i \geq d_k$$

找零问题 —— 动态规划

- 基本想法：先解决凑1元的问题，然后解决凑2元的问题，再之后解决凑3元的问题.....直至所需的数额
- 将每个结果都保存在数组中！

0	1	2	3	4	5	6	7	8	9	10	11	12	...
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	...

找零问题 —— 动态规划

- 基本想法：先解决凑1元的问题，然后解决凑2元的问题，再之后解决凑3元的问题.....直至所需的数额
- 将每个结果都保存在数组中！

0	1	2	3	4	5	6	7	8	9	10	11	12	...
													...

1, 5, 7, 10 情况

$$\min\{\bullet\} + 1$$

找零问题 —— 递推算法

Find (n)

1, 5, 7, 10 情况

1. $Find[0] \leftarrow 0$
2. **if** $i < 0$ **then** $Find[i] \leftarrow \infty$
 //事实上, 对 $Find[-1] \sim Find[-10]$ 赋初值 ∞ 即可
3. **for** $i = 1$ **to** n
4. $Find[i] \leftarrow \min \{$
 $Find[i - 1] + 1,$
 $Find[i - 5] + 1,$
 $Find[i - 7] + 1,$
 $Find[i - 10] + 1$
 $\}$

动态规划

Dynamic Programming

动态规划

- 动态规划（Dynamic Programming）是一种通用的算法设计技术
- 由美国数学家理查德·贝尔曼在20世纪50年代发明，用于解决最优化问题
- “Programming”指的是一种表格方法，而不是编写计算机代码

动态规划

- 动态规划（DP）用于解决各种离散优化问题
- 在这些问题中，可能有多个解
- 每个解都有一个值，我们希望找到一个具有最佳（最小或最大）值的解
- 我们称这种解为该问题的最佳解之一
 - 因为可能有多个最优解都达到了最优值

动态规划

- ① 刻画最优解的结构特性
 - $P(X)$
 - 例如 $P(n), P(n, w)$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

■ 通常而言， ϕ 是 $\max\{ \}$ 或者 $\min\{ \}$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

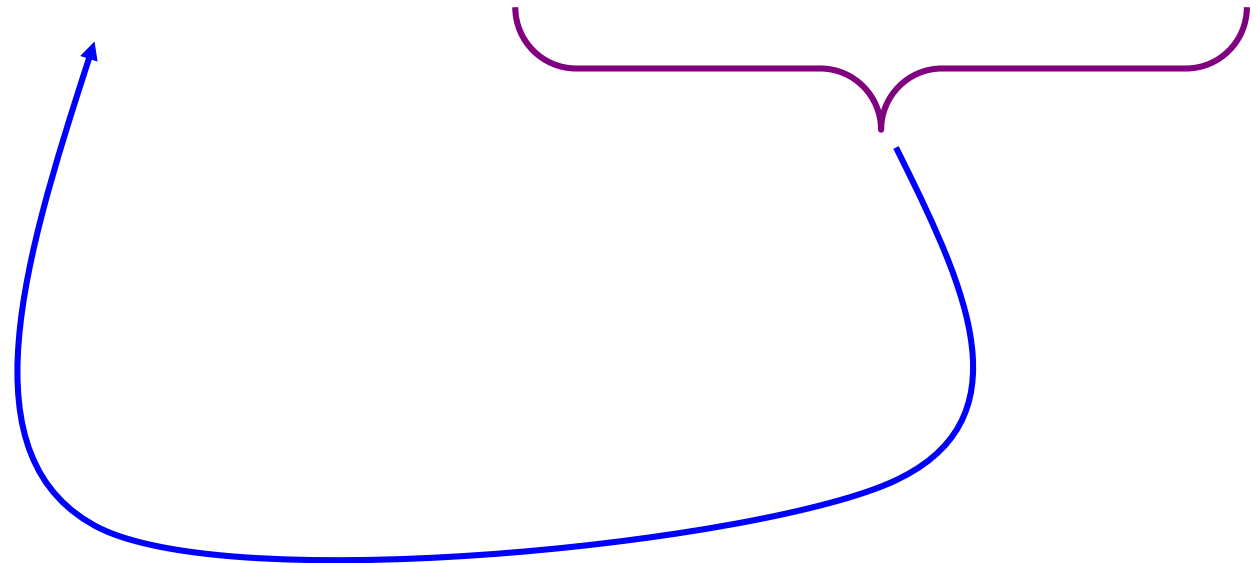
■ 示例

□ 斐波那契数: $F(n) = F(n-1) + F(n-2)$

□ 找零问题: $M(n) = \min \{ 1 + M(n-d_i) \}$

动态规划

■ ③ 自底而上计算

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$


■ ④ 注意初值

算法策略

- **贪婪策略** 逐步建立一个解决方案，每一步都“目光短浅地”地选择优化一些局部目标
- **分治策略** 将一个问题分解为多个不相交的子问题，独立解决每个子问题，并将子问题的解结合起来形成原问题的解
- **动态规划** 将一个问题分解成一系列相互存在重叠的子问题，并不断由子问题的解形成越来越大的问题的解

最长单调增子序列

Longest Increasing Subsequence

最长单调子序列

- 序列 S 的子序列 (subsequence) 是通过从 S 中删除零个或多个项并保持其余项的原有次序得到的
 - 例如, pred、sdn、predent都是“president”的子序列
 - 注意: 与子串是不一样的!

最长单调子序列

- 给定数值 x_1, x_2, \dots, x_n 构成的序列 S ，若其子序列 $x_{i_1} x_{i_2}, \dots, x_{i_k}$ ($i_{j+1} > i_j$) 对所有 j 都满足 $x_{i_{j+1}} > x_{i_j}$ ，则称该子序列是**单调的 (monotonic)**
- 希望找到 S 的一个**最长的**单调子序列
 - (可能不止一个)
- 该定义是单调增的子序列，也可以类似地定义单调减子序列、单调不增子序列、单调不减子序列

最长单调子序列

■ 例:

- 考察序列 1, 8, 2, 9, 3, 10, 4, 5
- 1, 2, 3, 4, 5 和 1, 8, 9, 10 都是它的单调增子序列
- 而其中 1, 2, 3, 4, 5 是最长的

■ 最短的单调增子序列长度为1

- 仅包含一项
- 也说明最长单调子序列必定存在，且长度至少为1

最长单调子序列

■ 算法思路

- 令 $S[1]S[2]S[3]...S[n]$ 表示输入序列
- 令 $L(i)$ ($1 \leq i \leq n$) 表示以 $S[i]$ 结束的最长单调递增子序列的长度
 - 即子序列的最后一项是 $S[i]$
 - 此时只需考虑前 i 项 $S[1]S[2]...S[i]$ 的以 $S[i]$ 结束的最长单调递增子序列即可
- 总目标为 $\max \{L(1), ..., L(n)\}$
 - (总要有有一个“最后一项”的)

最长单调子序列

■ 例: $S = 1, 8, 2, 9, 3, 10$

□ $L(6) = ?$

□ $L(6) = \max\{L(1), L(2), L(3), L(4), L(5)\} + 1$

■ 例: $S = 1, 8, 2, 9, 3$

□ $L(5) = ?$

□ $L(5) = \max\{L(1), L(2), L(3), L(4)\} + 1 ?$

□ $L(4) = 3$ —— “1, 2, 9”

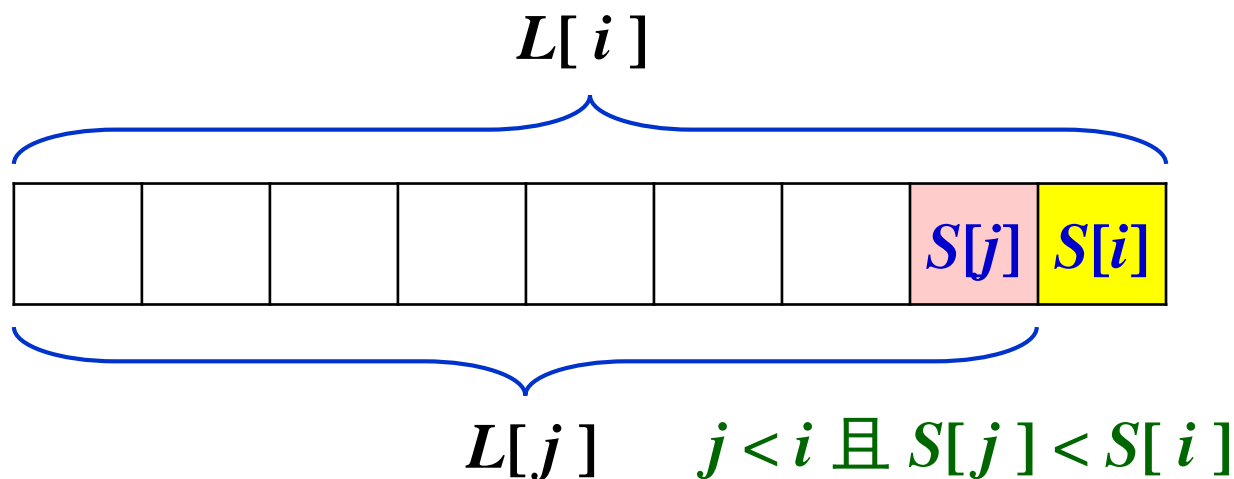
□ $L(5) = 3 + 1 ?$

□ $L(5) = \max\{L(1), L(3)\} + 1$

最长单调子序列

■ 计算 $L[i]$ 的递推式

$$L(i) = \begin{cases} \max_{1 \leq j < i \text{ and } S[j] < S[i]} L(j) + 1 & , \text{若存在 } j \text{ 使得} \\ & 1 \leq j < i \text{ 且 } S[j] < S[i] \\ 1 & , \text{否则} \end{cases}$$



最长单调子序列

输入：序列 $S[1]S[2]S[3]...S[n]$

输出：输入序列的最长单调递增子序列的长度 Len

1. **for** $i = 1$ **to** n **do**

等同于前述 “ $L(j) + 1 > L(i) + 1$ ” 情况

2. $L(i) \leftarrow 1$

3. **for** $j = 1$ **to** $i-1$ **do**

4. **if** $S[j] < S[i]$ **and** $L(j) \geq L(i)$ **then**

5. $L(i) \leftarrow L(j) + 1$

6. $Len \leftarrow \max \{L(1), ..., L(n)\}$

也可以
边算边更新

最长单调子序列

i	S	L[i]	
1	1	1	1
2	8	2	1 8
3	2	2	1 2
4	9	3	1 8 9
5	3	3	1 2 3
6	10	4	1 8 9 10
7	4	4	1 2 3 4
8	5	5	1 2 3 4 5

最长单调子序列

■ 算法思路

- 令 $S[1]S[2]S[3]...S[n]$ 表示输入序列
- 令 $L(i)$ ($1 \leq i \leq n$) 表示以 $S[i]$ 结束的最长单调递增子序列的长度（即子序列的最后一项是 $S[i]$ ）

■ 如果还要输出最长单调子序列（之一）的话——

- 令 $P(i)$ 表示这个最长单调递增子序列中在 $S[i]$ 之前一项（即倒数第二项）的位置
 - 如果 $L(i) = 1$ （即该子序列只包含 $S[i]$ ，而没有倒数第二项），则取 $P(i) = 0$
- 令 T 表示整个序列的最长单调递增子序列

最长单调子序列

■ 例: $S = 1, 8, 2, 9, 3, 10$

□ $L(6) = \max\{L(1), L(2), L(3), L(4), L(5)\} + 1$

□ $P(6) = 5$

■ 例: $S = 1, 8, 2, 9, 3$

□ $L(5) = \max\{L(1), L(3)\} + 1$

□ $P(5) = 3$

							$S[P(i)]$	$S[i]$
--	--	--	--	--	--	--	-----------	--------

i	S	L[i]	
1	1	1	1
2	8	2	1 8
3	2	2	1 2
4	9	3	1 8 9
5	3	3	1 2 3
6	10	4	1 8 9 10
7	4	4	1 2 3 4
8	5	5	1 2 3 4 5

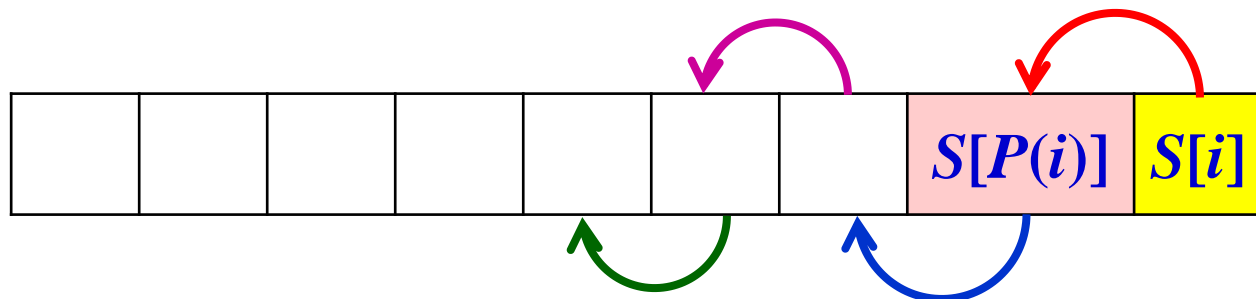
最长单调子序列

```
1. for  $i = 1$  to  $n$  do  
2.    $L(i) \leftarrow 1, P(i) \leftarrow 0$   
3.   for  $j = 1$  to  $i-1$  do  
4.     if  $S(j) < S(i)$  and  $L(j) \geq L(i)$  then  
5.        $L(i) \leftarrow L(j) + 1, P(i) \leftarrow j$   
6.  $Len \leftarrow \max \{L(1), \dots, L(n)\}$ 
```

最长单调子序列

■ 算法思路

- 令 $S[1]S[2]S[3]...S[n]$ 表示输入序列
- 令 $L(i)$ ($1 \leq i \leq n$) 表示以 $S[i]$ 结束的最长单调递增子序列的长度（即子序列的最后一项是 $S[i]$ ）
- 令 $P(i)$ 表示这个最长单调递增子序列中在 $S[i]$ 之前一项（即倒数第二项）的位置
 - 如果 $L(i) = 1$ （即该子序列只包含 $S[i]$ ），则取 $P(i) = 0$
- “记住你前一个人是谁” “没有前一个人的就是排头”



最长单调子序列

i	S	L[i]		P[i]
1	1	1	1	0
2	8	2	1 8	1
3	2	2	1 2	1
4	9	3	1 8 9	2
5	3	3	1 2 3	3
6	10	4	1 8 9 10	4
7	4	4	1 2 3 4	5
8	5	5	1 2 3 4 5	7



最长单调子序列

```
1. for  $i = 1$  to  $n$  do
2.    $L(i) \leftarrow 1, P(i) \leftarrow 0$ 
3.   for  $j = 1$  to  $i-1$  do
4.     if  $S[j] < S[i]$  and  $L(j) \geq L(i)$  then
5.        $L(i) \leftarrow L(j) + 1, P(i) \leftarrow j$ 
6.  $Len \leftarrow \max \{L(1), \dots, L(n)\}$  //假设最大值为  $L(k)$ 
7.  $i \leftarrow 1$  //回溯
8.  $j \leftarrow k$ 
9. do
10.   $T(i) \leftarrow S[j], i \leftarrow i + 1, j \leftarrow P(j)$ 
11. until  $j=0$ 
12. output  $Len$  及  $T$  的反序
```

$O(n^2)$

```
1. for  $i = L(k)$  downto 1
2.   print  $T(i)$ 
```

最大子段和问题

Maximum Sum Subarray Problem

最大子段和问题

- 由Ulf Grenander在1977年提出
- 给定由 n 个整数（可能为负整数）组成的序列 a_1, a_2, \dots, a_n ，求该序列的形如 $a_i + a_{i+1} + \dots + a_j$ 的和（称作子段和）中的最大非负值
 - 该子段和的长度为 $j - i + 1$
- 如果该序列所有子段和均是负整数时，定义其最大子段和为0
 - 可视作选择了长度为0的子段和

最大子段和问题

- 给定由 n 个整数（可能为负整数）组成的序列 a_1, a_2, \dots, a_n ，求该序列的形如 $a_i + a_{i+1} + \dots + a_j$ 的和（称作**子段和**）中的最大非负值

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

- 例：

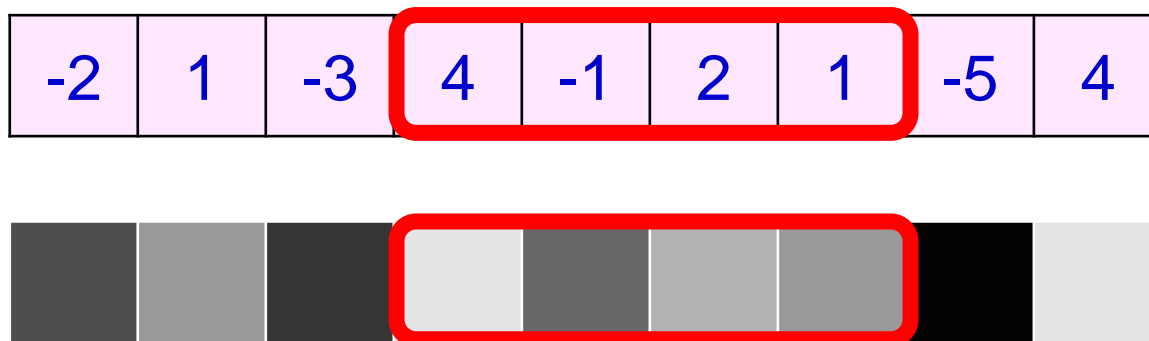
-2	1	-3	4	-1	2	1	-5	4
----	---	----	---	----	---	---	----	---

最大子段和问题

- 给定由 n 个整数（可能为负整数）组成的序列 a_1, a_2, \dots, a_n ，求该序列的形如 $a_i + a_{i+1} + \dots + a_j$ 的和（称作**子段和**）中的最大非负值

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

- 例：



最大子段和问题

- 蛮力法？
 - 共有约 $n^2/2$ 个不同的子段和
- Kadane算法

最大子段和问题

■ Kadane算法

- 令 $C(j)$ 表示必须以元素 a_j 结尾（因此长度至少为1）的最大子段和（允许为负数值）

- 即

$$C(j) = \max_{1 \leq i \leq j} \left\{ \sum_{k=i}^j a_k \right\}$$

- 目标则是计算

$$\max \left\{ 0, \max_{1 \leq j \leq n} \{ C(j) \} \right\}$$

最大子段和问题

■ Kadane算法

□ 令 $C(j)$ 表示必须以元素 a_j 结尾的最大子段和

□ 即

$$C(j) = \max_{1 \leq i \leq j} \left\{ \sum_{k=i}^j a_k \right\}$$

□ 此时就有两种情况：

- (1) 长度为1，即仅仅包含 a_j
- (2) 长度大于1

最大子段和问题

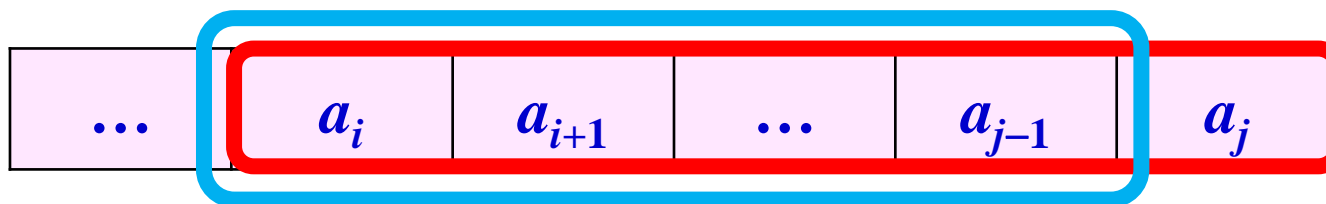
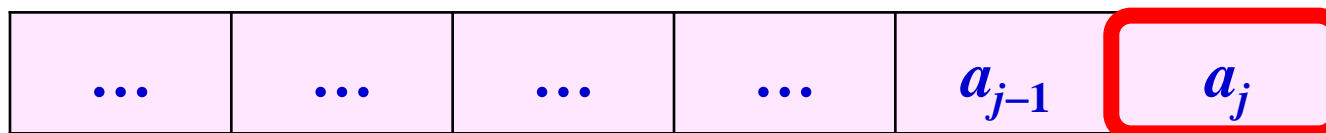
■ Kadane算法

□ 令 $C(j)$ 表示必须以元素 a_j 结尾的最大子段和

□ 有两种情况：

■ (1) 长度为1，即仅仅包含 a_j

■ (2) 长度大于1



$$C(j-1) > 0$$

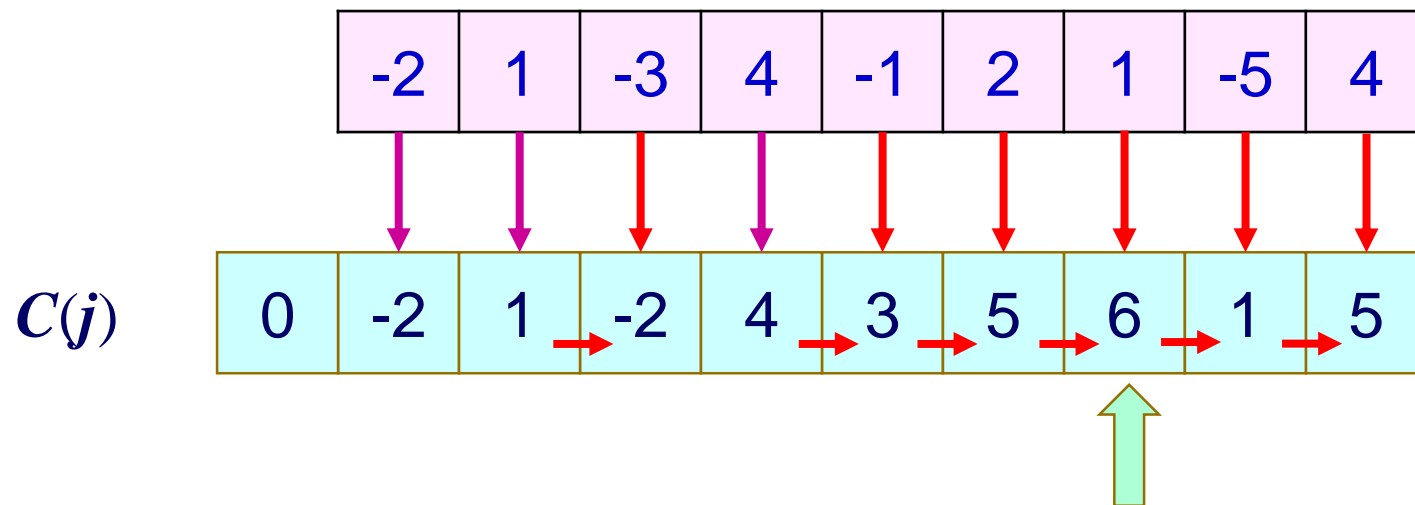
最大子段和问题

■ Kadane算法

□ 令 $C(j)$ 表示必须以元素 a_j 结尾的最大子段和

$$C(j) = \begin{cases} a_j + C(j-1) & \text{if } C(j-1) > 0 \\ a_j & \text{else} \end{cases}$$

■ 例:



最大子段和问题

Algorithm MaxSum (A, n)

输入：数组/序列 A ，长度 n

输出：诸 $C(j)$ 及最大子段和 sum

1. $C(0) \leftarrow 0$
2. **for** $j = 1$ **to** n **do**
3. $C(j) \leftarrow a_j$
4. **if** $C(j-1) > 0$ **then**
5. $C(j) \leftarrow C(j) + C(j-1)$
6. $sum \leftarrow \mathbf{max} \{ C(j) \}$ **for all** j

也可以
边算边更新

最大子段和问题

Algorithm MaxSum (A, n)

输入：数组/序列 A ，长度 n

输出：最大子段和

1. $current_sum \leftarrow 0$ //此即 $C[j]$
2. $best_sum \leftarrow 0$
3. **for** $j = 1$ **to** n **do**
4. **if** $current_sum > 0$ **then**
5. $current_sum \leftarrow current_sum + a_j$
6. **else** $current_sum \leftarrow a_j$
7. **if** $current_sum > best_sum$ **then**
8. $best_sum \leftarrow current_sum$
9. **return** $best_sum$

最大子段和问题

- 如果我们不仅要得到最大子段和
- 还希望知道取得最大子段和时的具体子段
- 那么应该怎么做？

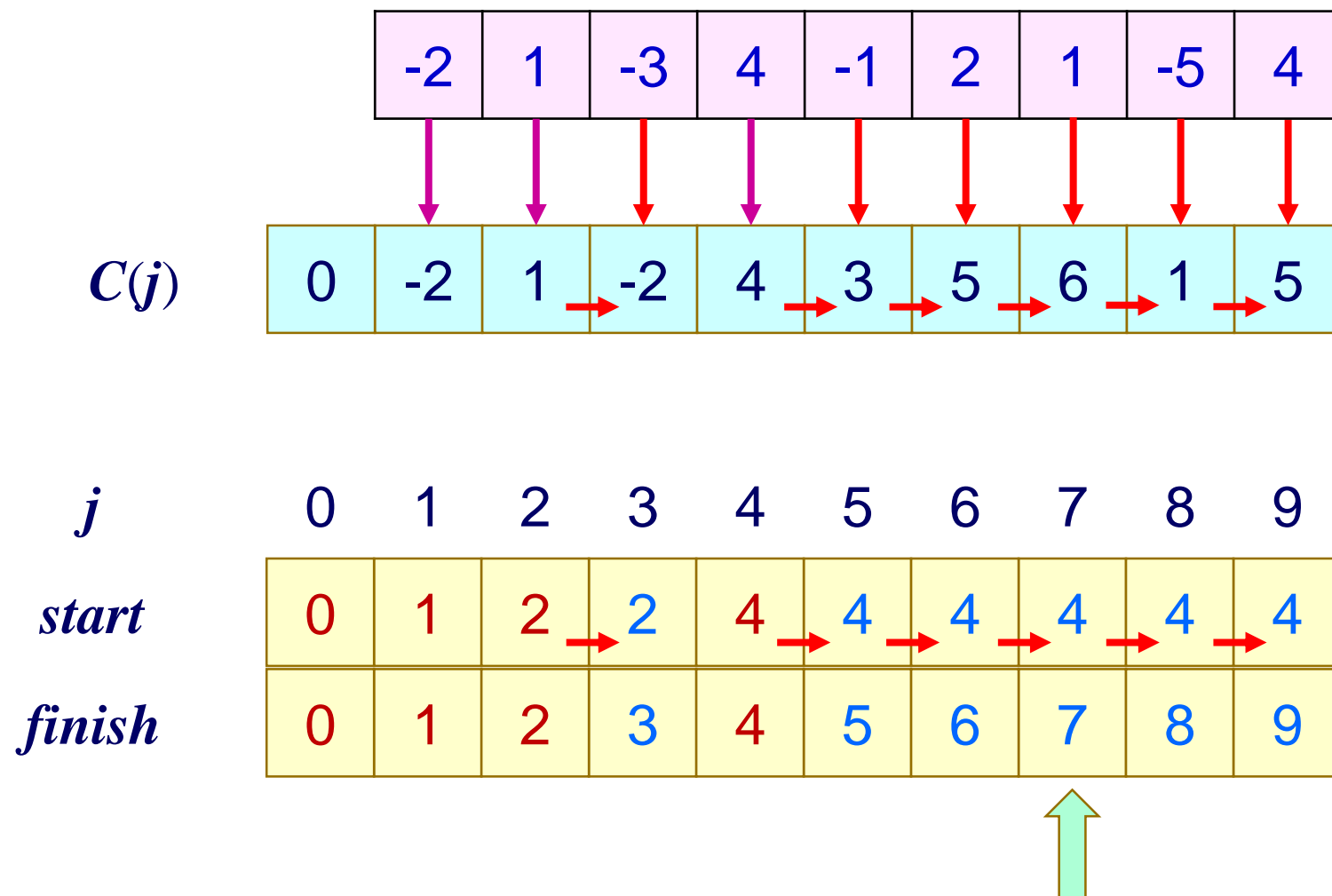
最大子段和问题

- 使用二元组 $(s(j), f(j))$ 记录取得 $C(j)$ 时的具体子段
 - 意为 $(start, finish)$

最大子段和问题

- 记录取得 $C(j)$ 时的具体子段
- 初始化为 $(s(0), f(0)) \leftarrow (0, 0)$
- Kadane算法
 - 令 $C(j)$ 表示必须包含元素 a_j 的最大子段和
 - 有两种情况:
 - (1) 长度为1, 即仅仅包含 a_j
 - (2) 长度大于1
 - 第一种情况: $(s(j), f(j)) \leftarrow (j, j)$
 - 第二种情况: $(s(j), f(j)) \leftarrow (s(j-1), j)$
- 具体算法由学生完成

最大子段和问题



0-1 背包问题

0-1 Knapsack Problem

背包问题

■ 背包问题是一个典型的组合优化问题：

- 假设共有 n 种物品，其中第 i 种物品的价值为 v_i ，重量为 w_i
 - 假定 v_i 和 w_i 都是整数
- 确定要从这 n 种物品中选择哪些种、每种选择多少，将其装入背包，使得这些物品的总重量不超过给定的限制 W ，并且总价值尽可能大

■ 0-1背包问题

- 对于每种物品，当在考虑是否将其装入背包时，要么全部装入背包，要么全部不装入背包，而不能只装入物品 i 的一部分

■ 注意：凡论及“背包问题”时，将“体积”和“重量”视作同一个属性，“收益”和“价值”含义相同

0-1背包问题

■ 贪婪策略之一：

- 按照 v_i/w_i 的降序依次考虑各个物品
- 在右例中，将选取物品 {5, 2, 1}，取得总价值 35

W = 11

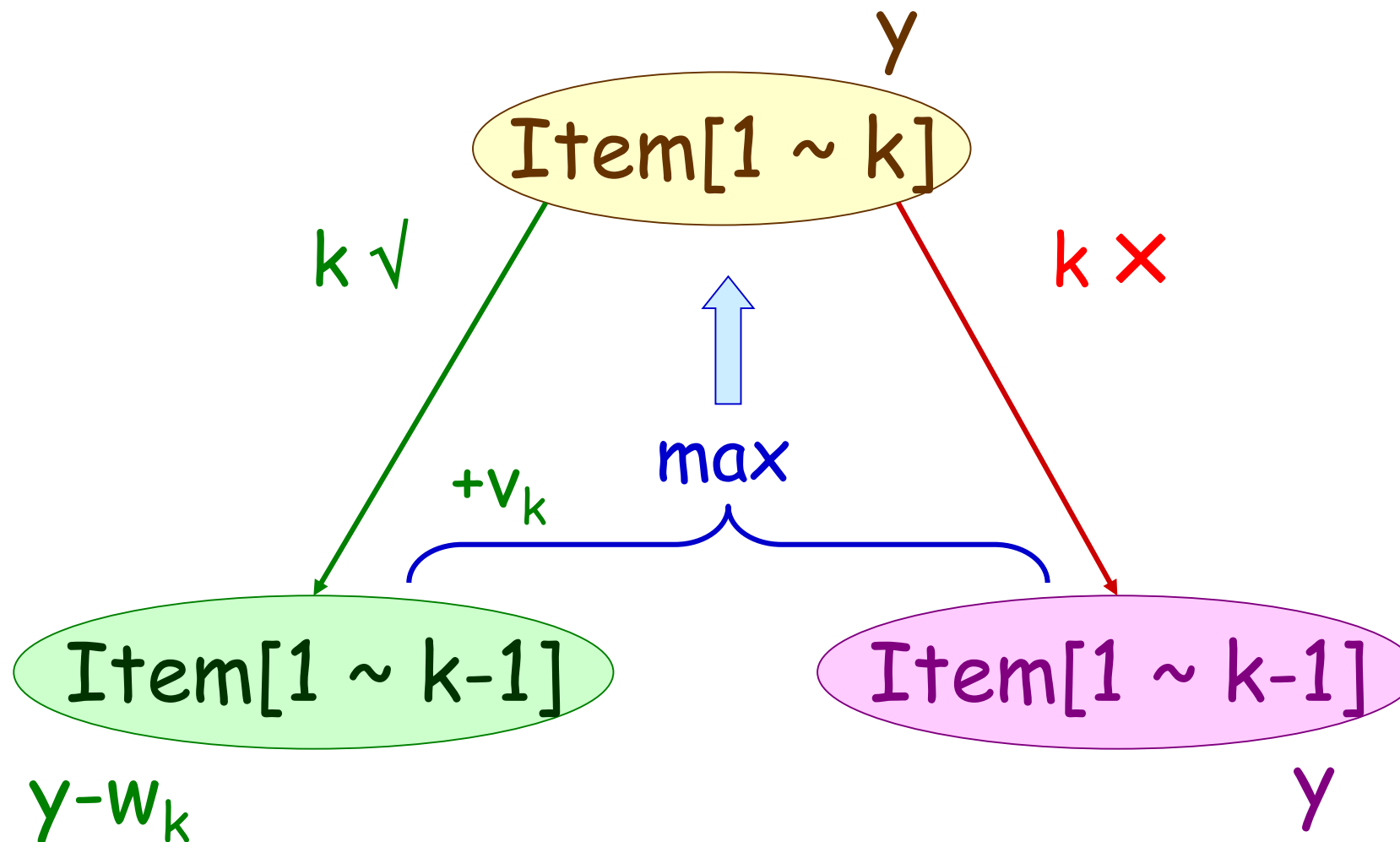
Item	Value	Weight	V/W
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.67
5	28	7	4

然而，选取物品 { 4, 3 }，取得总价值 40

0-1背包问题

- 考虑下述子问题 $P(k, y)$ ——只使用前 k 种物品，总重不超过 y
 - 令 $F(k, y)$ 表示仅使用前 k 种物品且背包容量为 y 时的最优解的总价值
 - 即子问题 $P(k, y)$ 问题的最优解
 - 原问题的目标为 $F(n, W)$

0-1背包问题



0-1背包问题

■ 例:

$$W = 11$$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

■ $F(5, 11) =$

$$\text{Max}\{ 28 + F(4, 11-7), F(4, 11) \}$$

0-1背包问题

- 令 $F(k, y)$ 表示仅使用前 k 种物品且背包容量为 y 时的最优解的总价值
 - 情形一：不选取物品 k
 - 在总重量限制 y 下从物品 $\{1, 2, \dots, k-1\}$ 中选择最优方案
 - 情形二：选取物品 k
 - 需要满足 $w_k \leq y$
 - 已使用重量 w_k ，新的总重量限制变为 $y - w_k$
 - 在总重量限制 $y - w_k$ 下从物品 $\{1, 2, \dots, k-1\}$ 中选择最优方案

$$F(k, y) = \begin{cases} 0 & \text{if } k = 0 \text{ or } y = 0 \\ F(k-1, y) & \text{if } w_k > y \\ \max \{ F(k-1, y), v_k + F(k-1, y - w_k) \} & \text{otherwise} \end{cases}$$

0-1背包问题

输入: $n, w_1, \dots, w_n, v_1, \dots, v_n, W$

时间复杂度: $O(nW)$

1. **for** $y = 0$ **to** W
2. $F(0, y) \leftarrow 0$
3. **for** $k = 1$ **to** n
4. $F(k, 0) \leftarrow 0$
5. **for** $k = 1$ **to** n
6. **for** $y = 1$ **to** W
7. **if** ($w_k > y$)
8. $F(k, y) \leftarrow F(k - 1, y)$
9. **else**
10. $F(k, y) \leftarrow \max \{ F(k - 1, y), v_k + F(k - 1, y - w_k) \}$
11. **return** $F(n, W)$

0-1背包问题

		$W + 1$											
		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	{1}	0	1	1	1	1	1	1	1	1	1	1	1
	{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
	{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
	{1, 2, 3, 4}	0	1	6	7	7	18	22	+28	8	29	29	40
	{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

0-1背包问题

	$W + 1$											
	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	+18	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	+22	5	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

$n + 1$

最优方案: { 4, 3 }
 最优值 = 22 + 18 = 40

时间复杂度和
 空间复杂度: $O(nW)$

$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

背包问题

- 可以将时间复杂度和空间复杂度进一步降低，可参看：

<https://zhuanlan.zhihu.com/p/30959069>

投资问题



投资问题

- 有 m 元钱， n 项可能的投资项目
- $f_i(x)$ 表示将 x 元投入第 i 个项目能获得的收益
 - x 是非负整数
 - $f_i(x)$ 值非负
 - $f_i(x)$ 关于 x 是不减函数
- 目标是将投资的收益最大化
- 即 $\max \{ f_1(x_1) + f_2(x_2) + \dots + f_n(x_n) \}$
 $\text{s.t. } x_1 + x_2 + \dots + x_n = m$

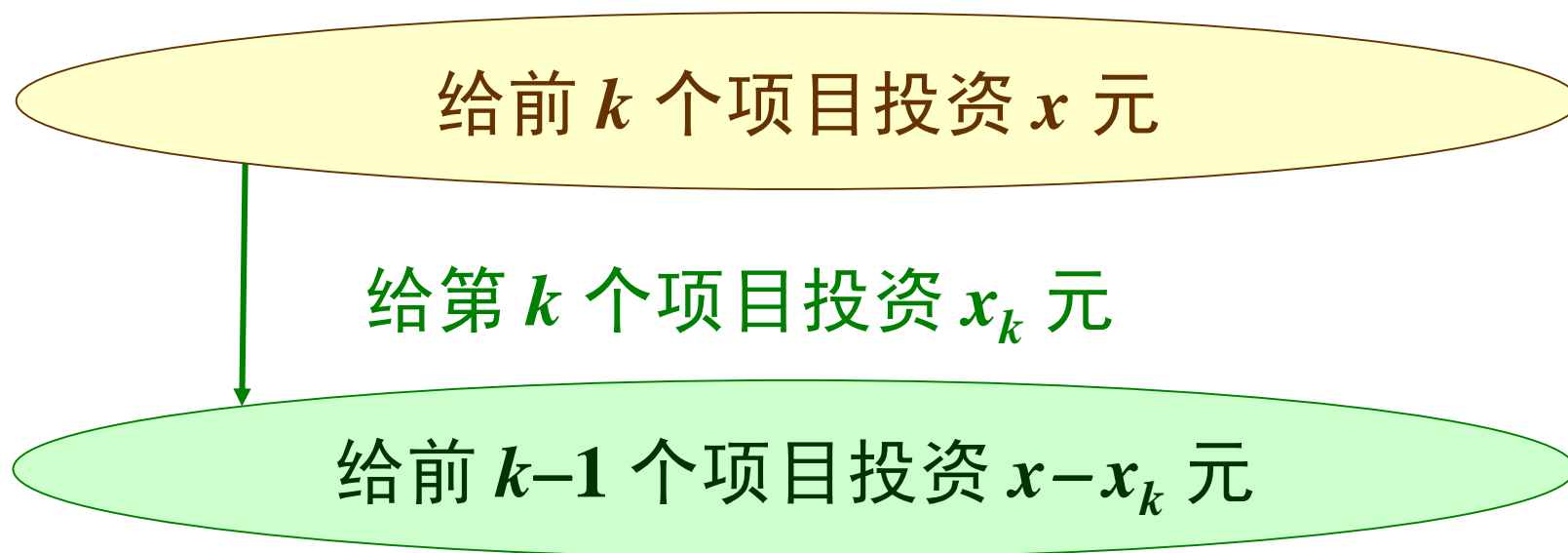
投资问题

■ 实例：5元，4项投资项目

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

投资问题

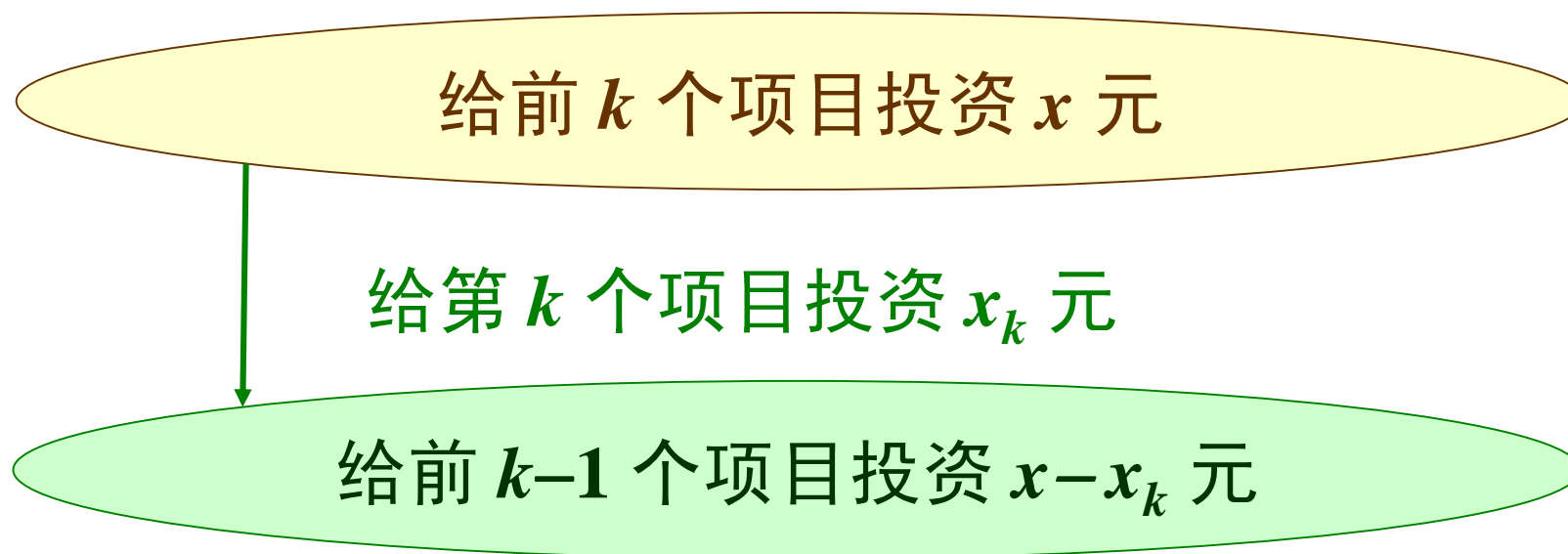
- 用 $F_k(x)$ 表示对前 k 个项目进行共计 x 元的投资所能取得的最大收益
- $x_k(x)$ 表示在 $F_k(x)$ 中投资给项目 k 的钱数
- 考虑“**最后一个**”项目



投资问题

$$F_k(x) = \max_{0 \leq x_k \leq x} \{f_k(x_k) + F_{k-1}(x - x_k)\}, 0 \leq x \leq m, 2 \leq k \leq n$$

$$F_1(x) = f_1(x), 0 \leq x \leq m$$



投资问题

$$F_k(x) = \max_{0 \leq x_k \leq x} \{f_k(x_k) + F_{k-1}(x - x_k)\}, 0 \leq x \leq m, 2 \leq k \leq n$$

$$F_1(x) = f_1(x), 0 \leq x \leq m$$

动态规划表

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)	x	f ₄ (x)
0	0	0	0	0	0	0			0	0
1	11	1	11	0	11	0			1	20
2	12	2	12	0	13	1			2	21
3	13	3	16	2	30	3			3	22
4	14	4	21	3	41	3			4	23
5	15	5	26	4	43	4			5	24

解: $F_4(5) = 61$

投资问题

$$F_k(x) = \max_{0 \leq x_k \leq x} \{f_k(x_k) + F_{k-1}(x - x_k)\}, 0 \leq x \leq m, 2 \leq k \leq n$$

$$F_1(x) = f_1(x), 0 \leq x \leq m$$

动态规划表

11

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)
0	0 0	0 0	0 0	0 0
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

0+11

30+11

20+41

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

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

投资问题

1. **for** $y = 1$ **to** m
2. $F_1(y) \leftarrow f_1(y)$
3. **for** $k = 2$ **to** n
4. **for** $y = 1$ **to** m
5. $F_k(y) \leftarrow \mathbf{max}_{0 \leq x_k \leq y} \{ f_k(x_k) + F_{k-1}(y-x_k) \}$
6. **return** $F_n(m)$

投资问题

■ 时间复杂度

- 计算 $F_k(y)$ ($2 \leq k \leq n, 1 \leq y \leq m$) 时候需要 $y+1$ 次加法和 y 次比较

加法次数	$\sum_{k=2}^n \sum_{y=1}^m (y+1) = \frac{1}{2}(n-1)m(m+3)$
比较次数	$\sum_{k=2}^n \sum_{y=1}^m y = \frac{1}{2}(n-1)m(m+1)$
时间复杂度	$O(nm^2)$

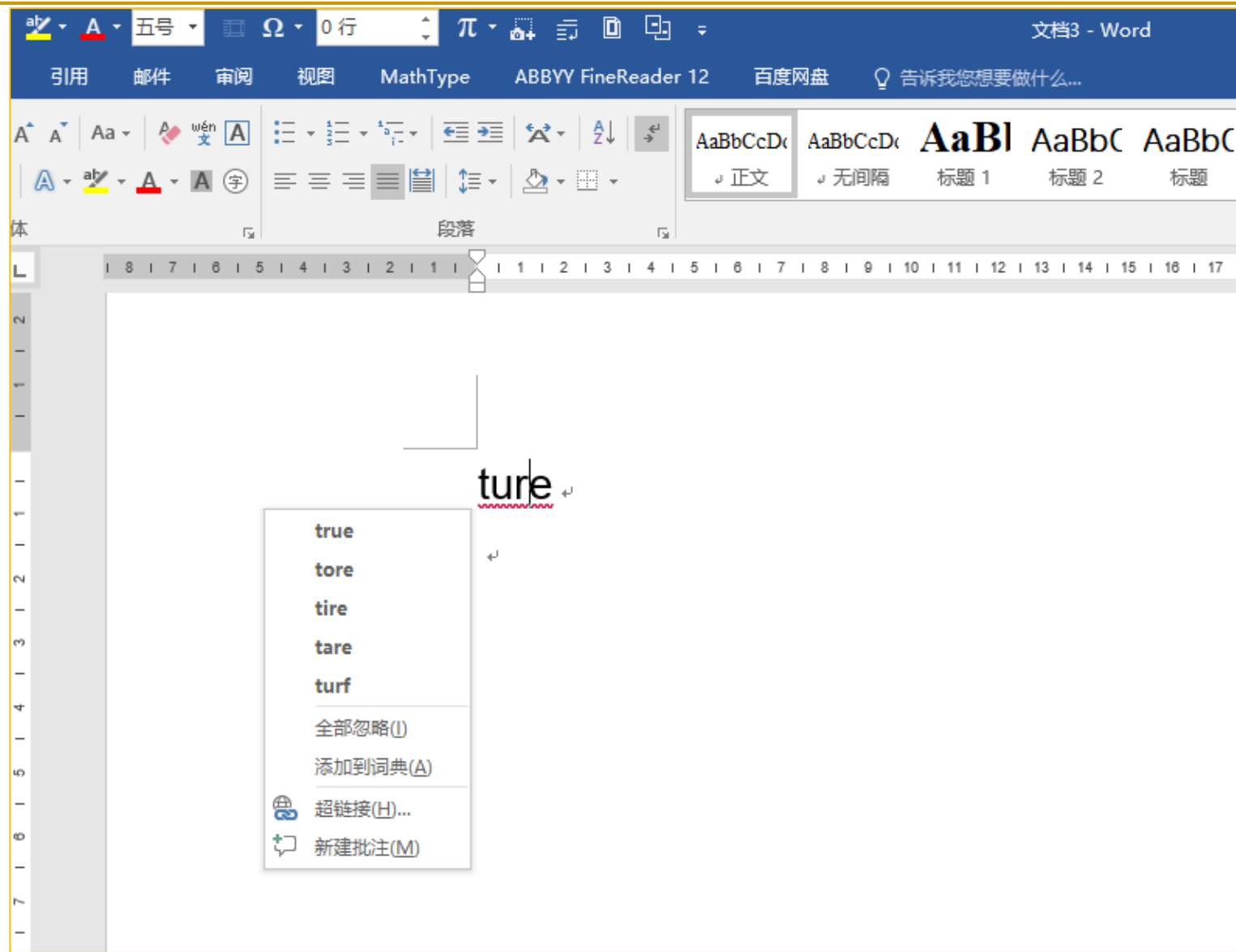
序列的比较

Comparison of Two Sequences

刘铎

liuduo@bjtu.edu.cn

序列的比较



序列比较

- 最长公共子序列

Longest Common Subsequence

- 最短公共超序列

Shortest Common Supersequence

- 编辑距离

Edit Distance Between Two Sequences

最长公共子序列

Longest Common Subsequence (LCS)

刘铎

liuduo@bjtu.edu.cn

a	b	c	d	d	e	a	c	d	e
a	b	c	a	d	c	a	b	b	e

最长公共子序列

- 序列 S 的**子序列** (subsequence) 是通过从 S 中删除零个或多个项并保持其余项的原有次序得到的
 - 例如, pred、sdn、predent都是“president”的子序列
 - 注意: 与**子串**是不一样的!
- **最长公共子序列问题**指的是在两个序列之间找到一个具有**最大**长度的公共子序列
 - 序列的长度指的是序列的项数
 - 空序列是任意两个序列的公共子序列
 - 任一个序列和空序列的最长公共子序列都是空序列

最长公共子序列

■ 定义1:

- 给定一个序列 $X = x_1x_2...x_m$ ，另一个序列 $Z = z_1z_2...z_k$ 称作是 X 的**子序列**，指的是存在关于 X 的指数的严格增的序列 $i_1i_2...i_k$ ，那么对于所有的 $j = 1, 2, ..., k$ ，我们有 $x_{i_j} = z_j$

■ 例1:

- 若 $X = \text{abcdefg}$ ，则 $Z = \text{abdg}$ 是 X 的一个子序列

$X = \text{a b c d e f g},$

$Z = \text{a b } \quad \text{d} \quad \text{g}$

最长公共子序列

■ 定义2:

- 给定两个序列 X 和 Y ，如果 Z 既是 X 的子序列也是 Y 的子序列，则称 Z 是 X 和 Y 的公共子序列（common subsequence）

■ 例2:

- 设 $X = abcdefg$ ， $Y = aadgfd$ ，则 $Z = adf$ 是 X 和 Y 的公共子序列

$X = a b c \quad d \quad e f g$

$Y = a \quad a d g \quad f \quad d$

$Z = a \quad \quad d \quad \quad f$

最长公共子序列

■ 定义3:

- X 和 Y 的**最长公共子序列** (Longest Common Subsequence, LCS) 指的是 X 和 Y 的公共子序列中具有最大长度者
- 序列的长度指的是序列中的项数/字母数
- 最长公共子序列**可能并不唯一**

■ 例3:

$X = \text{acbd}$

$Y = \text{abcd}$

acd 和 abd 都是 X 和 Y 的LCS

最长公共子序列

■ 示例一：

- 序列1: president
- 序列2: providence
- 其（唯一的）LCS为: priden

■ 示例二：

- 序列1: algorithm
- 序列2: alignment
- 二者的LCS之一为: algm

最长公共子序列问题

■ 应用：

□ 度量两个序列的相似程度

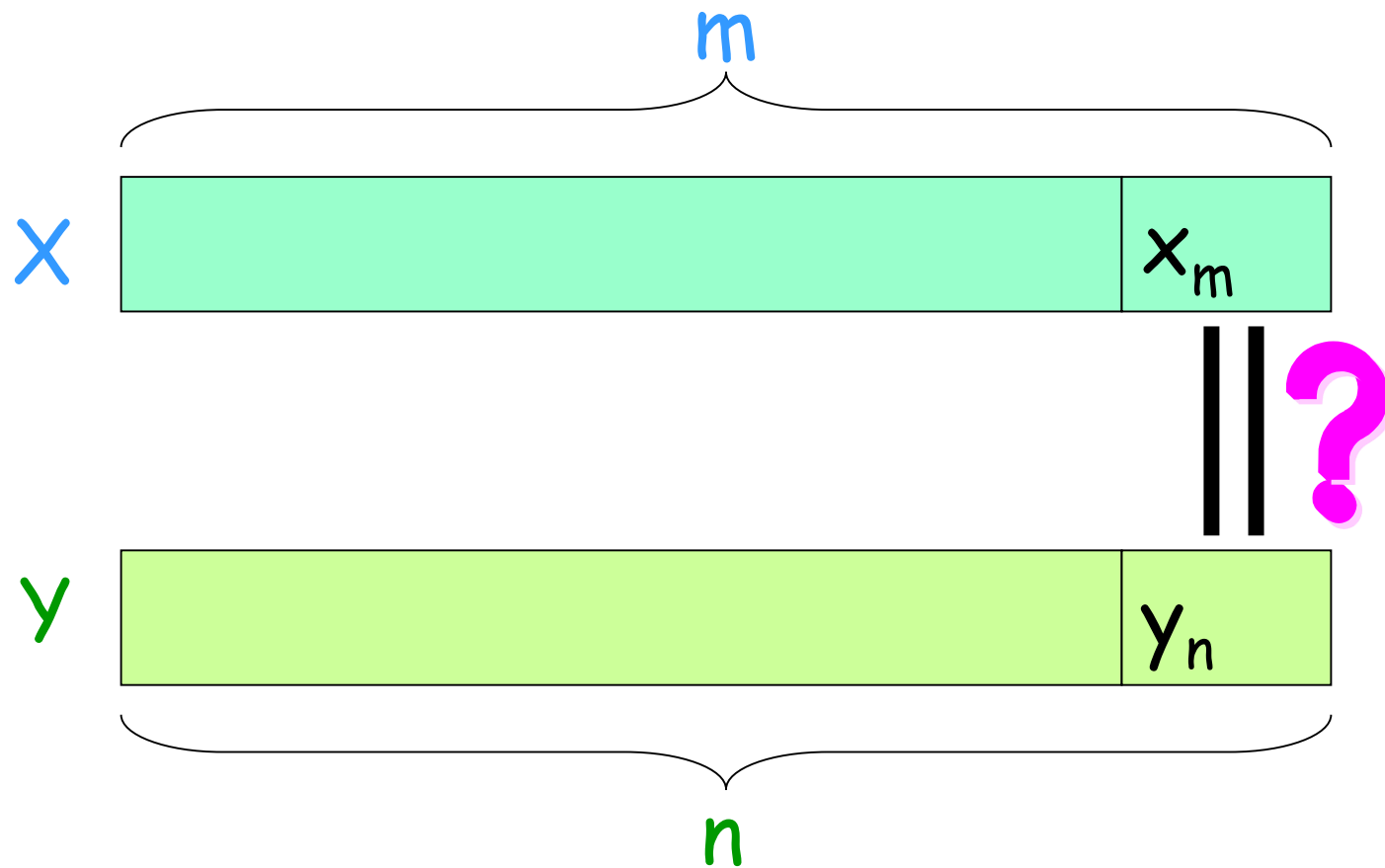
■ 例如 X : a b c d a c e; Y : b a d c a b e

■ LCS : b d a e, 长度为4

a	b	c		d		a	c		e
	b		a	d	c	a		b	e

□ Unix命令“diff”即使用LCS编辑距离作为度量

最长公共子序列的刻画



最长公共子序列的刻画

■ 定理（LCS的最优子结构）

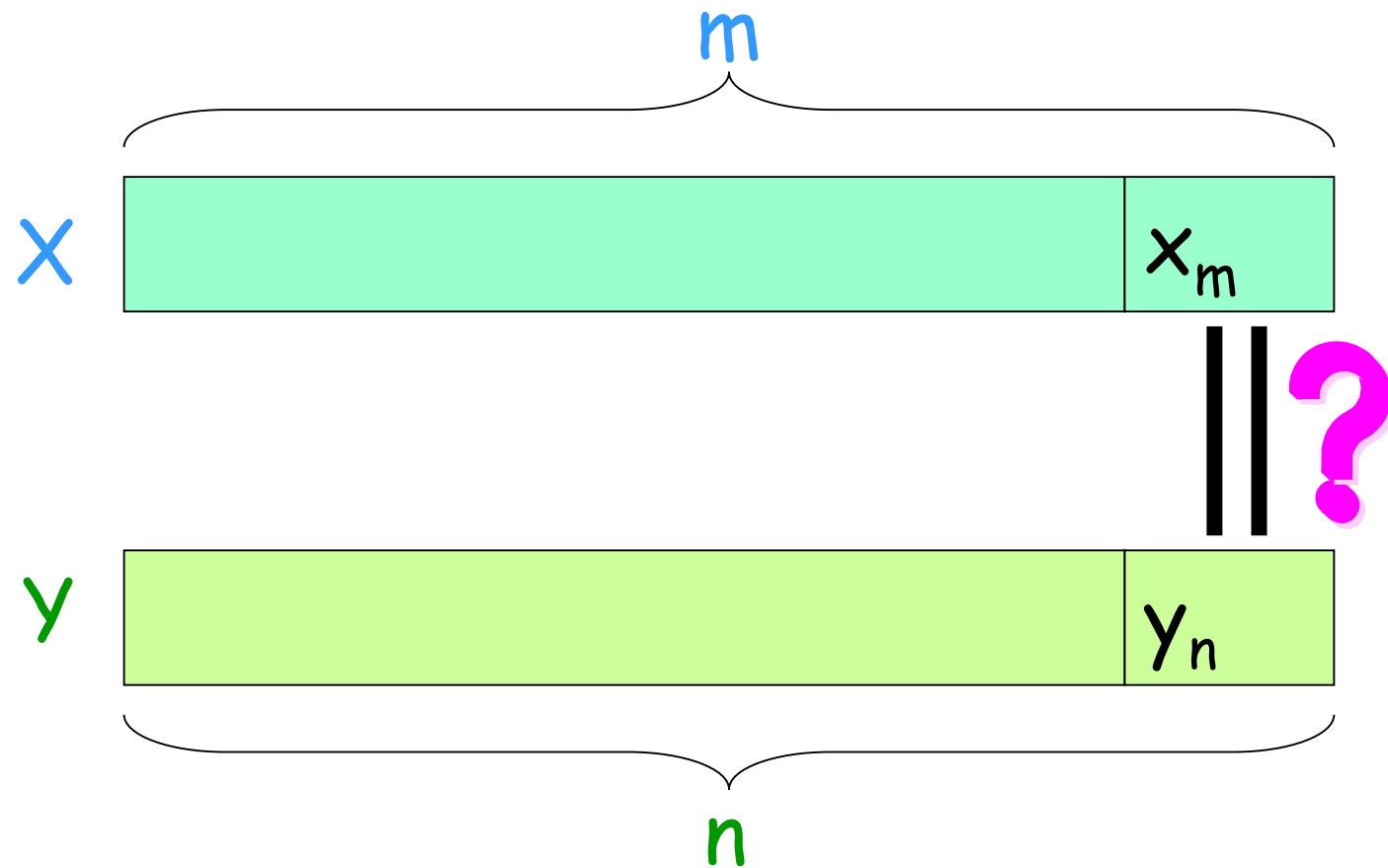
□ 设序列 $X = x_1x_2\dots x_m$, $Y = y_1y_2\dots y_n$,
 $Z = z_1z_2\dots z_k$ 是 X 和 Y 的任一个LCS

使用反证法,
否则必然可以在 Z 的末尾
添加 $x_m = y_n$

- 1. 若 $x_m = y_n$, 则 $z_k = x_m = y_n$, 且 $Z[1\dots k-1]$ 是 $X[1\dots m-1]$ 和 $Y[1\dots n-1]$ 的（一个）LCS
- 2-1. 若 $x_m \neq y_n$, 则 $z_k \neq x_m$ 蕴涵 Z 是 $X[1\dots m-1]$ 和 Y 的（一个）LCS
- 2-2. 若 $x_m \neq y_n$, 则 $z_k \neq y_n$ 蕴涵 Z 是 X 和 $Y[1\dots n-1]$ 的（一个）LCS

若 $(z_k = x_m \text{ 且 } z_k = y_n)$ 不成立
则必然有 $(z_k \neq x_m \text{ 或 } z_k \neq y_n)$ 成立

最长公共子序列的刻画



最长公共子序列

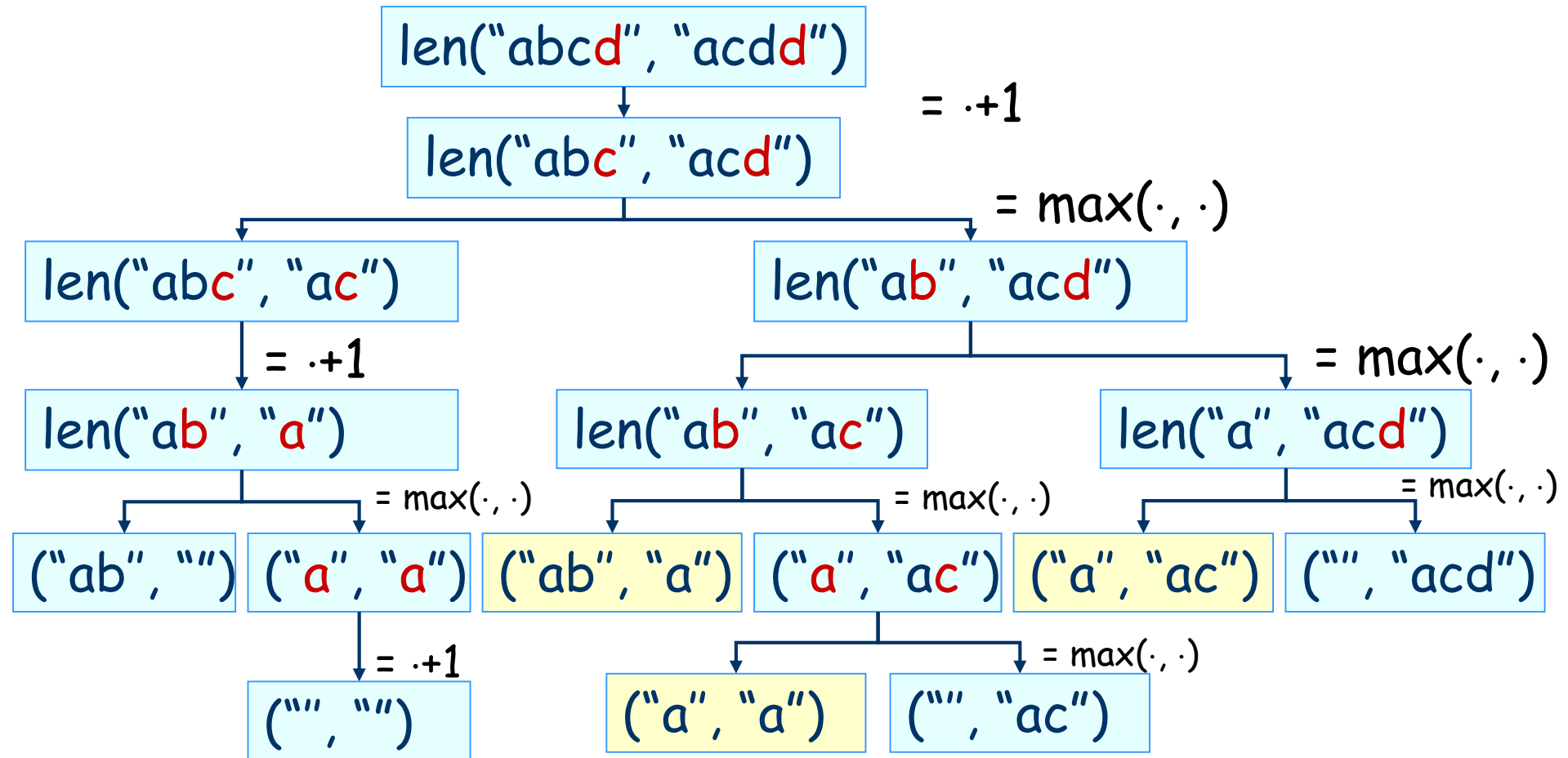
- 令 $len(i, j)$ 表示 $X[1...i]$ 和 $Y[1...j]$ 的LCS的长度
 $0 \leq i \leq m, 0 \leq j \leq n$



- **if** $X[i] = Y[j]$
 then return $len(i - 1, j - 1) + 1$
 else return $\max\{ len(i - 1, j), len(i, j - 1) \}$
- 该递归算法非常慢

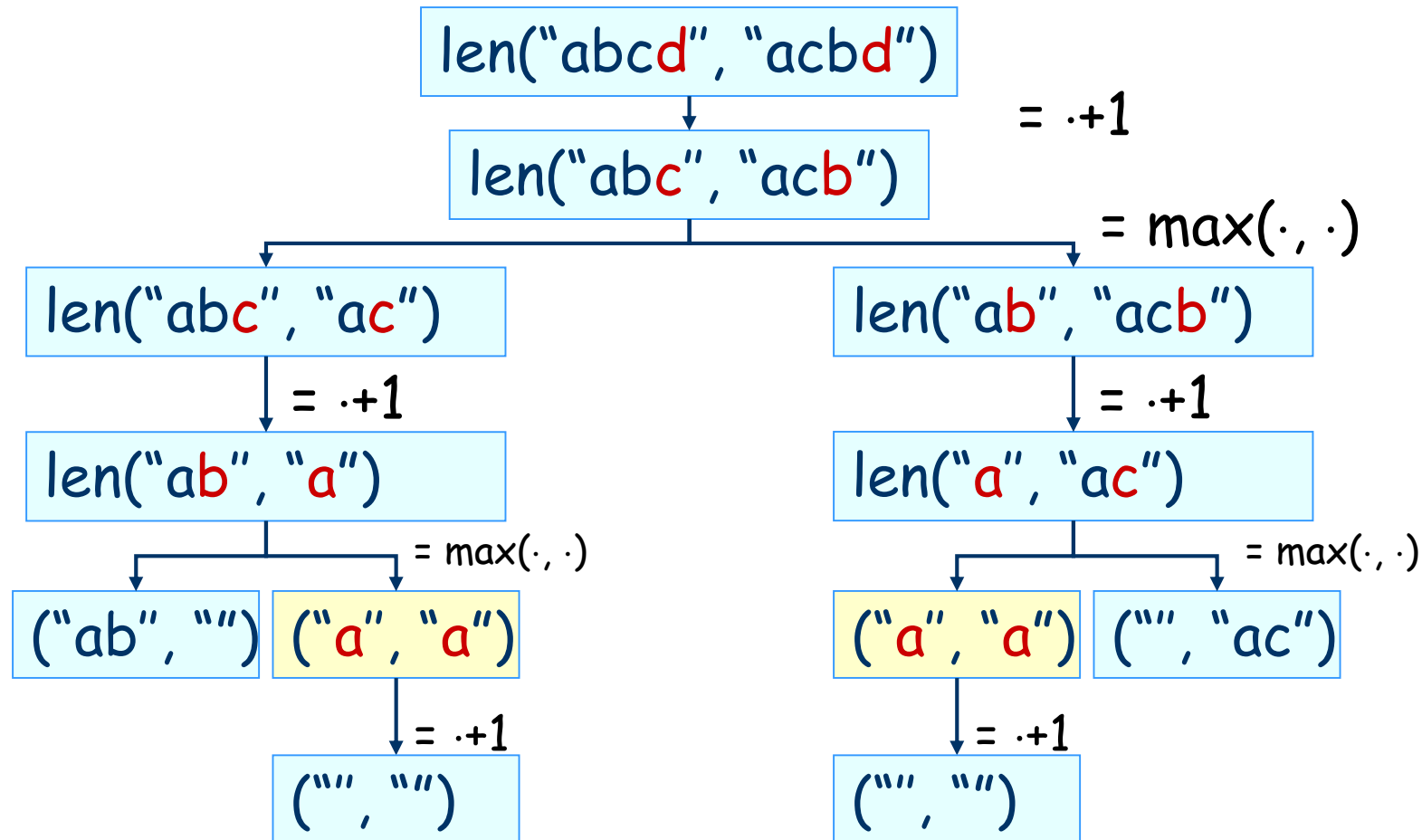
最长公共子序列

- $X[1..4] = \text{"abcd"}$, $Y[1..4] = \text{"acdd"}$



最长公共子序列

- $X[1..4] = \text{"abcd"}$, $Y[1..4] = \text{"acbd"}$

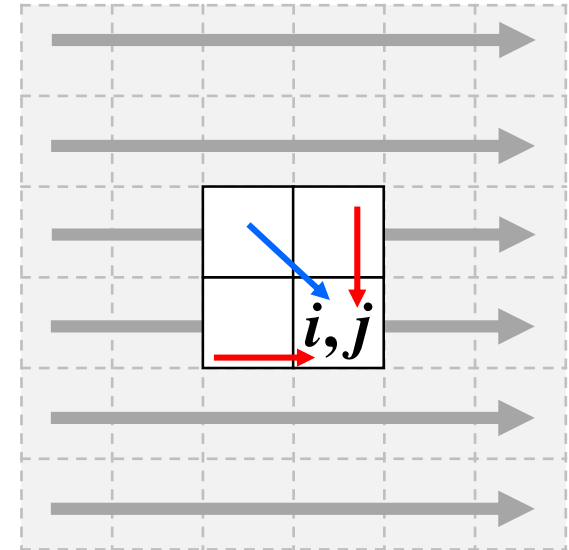


最长公共子序列

- 令 $len(i, j)$ 表示 $X[1...i]$ 和 $Y[1...j]$ 的LCS的长度
 $0 \leq i \leq m, 0 \leq j \leq n$



- **if** $X[i] = Y[j]$
 then return $len(i - 1, j - 1) + 1$
 else return $\max\{ len(i - 1, j), len(i, j - 1) \}$
- 该递归算法非常慢
- 所以考虑按特定的次序计算这 $m \times n$ 个 $len(i, j)$



计算LCS长度的算法

```
1.  for  $i = 1$  to  $m$ 
2.       $len(i, 0) \leftarrow 0$ 
3.  for  $j = 0$  to  $n$ 
4.       $len(0, j) \leftarrow 0$ 
5.  for  $i = 1$  to  $m$ 
6.      for  $j = 1$  to  $n$ 
7.          if  $X[i] = Y[j]$  then
8.               $len(i, j) \leftarrow len(i - 1, j - 1) + 1$ 
9.          else if  $len(i - 1, j) \geq len(i, j - 1)$  then
10.               $len(i, j) \leftarrow len(i - 1, j)$ 
11.          else  $len(i, j) \leftarrow len(i, j - 1)$ 
12. return  $len(m, n)$ 
```

最长公共子序列

■ 示例: $X[1..4] = \text{acbd}$, $Y[1..4] = \text{abcd}$

len(i,j)	j=0		1 a		2 b		3 c		4 d	
i=0	0	""	0	""	0	""	0	""	0	""
1 a	0	""	1	a	1	a	1	a	1	a
2 c	0	""	1	a	1	a	2	ac	2	ac
3 b	0	""	1	a	2	ab	2	ab	2	ab
4 d	0	""	1	a	2	ab	2	ab	3	abd

计算LCS (之一) 及其长度的算法

```
1. for  $i = 1$  to  $m$  do  $len(i, 0) \leftarrow 0$ 
2. for  $j = 0$  to  $n$  do  $len(0, j) \leftarrow 0$ 
3. for  $i = 1$  to  $m$  do
4.   for  $j = 1$  to  $n$  do
5.     if  $X[i] = Y[j]$  then
6.        $len(i, j) \leftarrow len(i - 1, j - 1) + 1$ 
7.        $b(i, j) \leftarrow 1$  // ↖
8.     else if  $len(i - 1, j) \geq len(i, j - 1)$  then
9.        $len(i, j) \leftarrow len(i - 1, j)$ 
10.       $b(i, j) \leftarrow 2$  // ↑
11.    else  $len(i, j) \leftarrow len(i, j - 1)$ 
12.       $b(i, j) \leftarrow 3$  // ←
13. return  $len(m, n)$  及  $b$ 
```

时间复杂度: $O(mn)$

计算LCS (之一) 的算法

- 示例: $X = a, b, c, d, a, c, e$; $Y = b, a, d, c, a, b, e$

len	i=0	1 a	2 b	3 c	4 d	5 a	6 c	7 e
j=0	0	0	0	0	0	0	0	0
1 b	0	0 ↑	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←
2 a	0	1 ↖	1 ↑	1 ↑	1 ↑	2 ↖	2 ←	2 ←
3 d	0	1 ↑	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑
4 c	0	1 ↑	1 ↑	2 ↖	2 ↑	2 ↑	3 ↖	3 ←
5 a	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	3 ↑
6 b	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑	3 ↑
7 e	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖

时间复杂度与空间复杂度: $O(mn)$

(回溯) 构造LCS的算法

- 可以使用 $b(i, j)$ 找到具体的LCS
- 从 $b(m, n)$ 开始, 回溯到某个 $b(0, j)$ 或 $b(i, 0)$ 为止

```
1.  $i \leftarrow m, j \leftarrow n, k \leftarrow 1$ 
2. while (  $i \neq 0$  and  $j \neq 0$  )
3.     if  $b(i, j) = 1$  then                                //↖
4.          $i \leftarrow i - 1, j \leftarrow j - 1$ 
5.          $LCS[k] \leftarrow X[i], k \leftarrow k + 1$ 
6.     if  $b(i, j) = 2$  then  $i \leftarrow i - 1$               //↑
7.     if  $b(i, j) = 3$  then  $j \leftarrow j - 1$               //←
8. for  $k = \text{len}(m, n)$  downto 1
9.     output  $LCS[k]$ 
```

(回溯) 构造LCS的算法

- 例: X: a,b,c,d,a,c,e; Y: b,a,d,c,a,b,e

len	0	1	2	3	4	5	6	7
		a	b	c	d	a	c	e
0	0	0	0	0	0	0	0	0
1 b	0	0	1	1	1	1	1	1
2 a	0	1	1	1	1	2	2	2
3 d	0	1	1	1	2	2	2	2
4 c	0	1	1	2	2	2	3	3
5 a	0	1	1	2	2	3	3	3
6 b	0	1	2	2	2	3	3	3
7 e	0	1	2	2	2	2	3	4

Diagram illustrating the backtracking path for the longest common subsequence (LCS) between X = a,b,c,d,a,c,e and Y = b,a,d,c,a,b,e. The path is highlighted with a green line and pink circles, starting from the bottom-right cell (7,7) and moving up and left to the top-left cell (0,0). The path consists of the following cells: (7,7), (6,7), (5,7), (4,7), (3,7), (2,7), (1,7), (0,7), (0,6), (0,5), (0,4), (0,3), (0,2), (0,1), (0,0).

b

a

c

e



最短公共超序列

Shortest Common Supersequence (SCS)

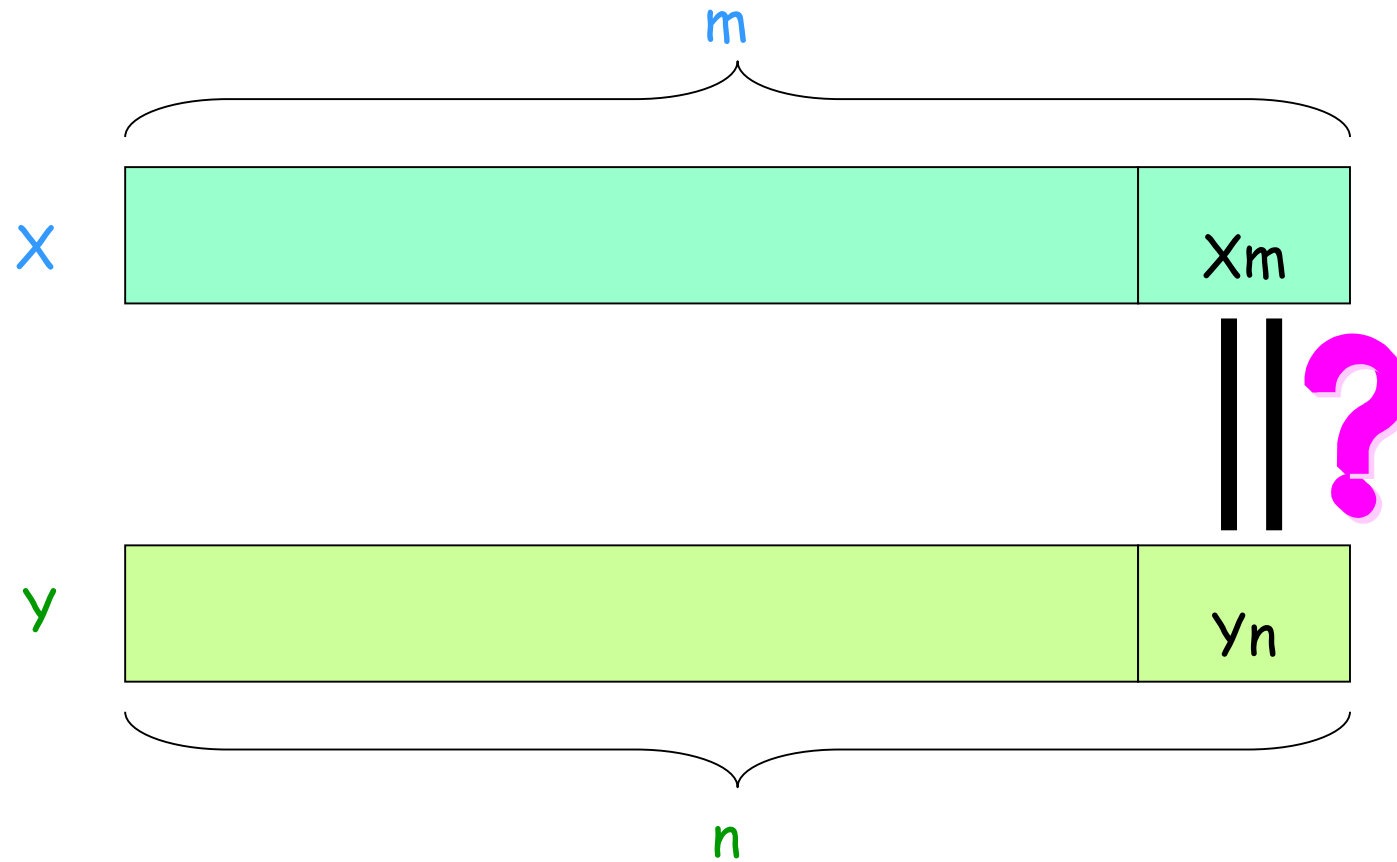
刘铎

liuduo@bjtu.edu.cn

最短公共超序列

- **定义：** 设 X 和 Y 是两个序列。如果 X 和 Y 都是 Z 的子序列，则称序列 Z 是 X 和 Y 的**公共超序列**（**common supersequence**）
- X 和 Y 的一个具有最短长度的公共超序列称作 X 和 Y 的一个**最短公共超序列**（**Shortest Common Supersequence**）
- **示例：** 设 $X = abc$ ， $Y = abb$ ，则 $abbc$ 和 $abcb$ 都是 X 和 Y 的最短公共超序列
- 任一个序列和空序列的最短公共超序列都是该序列自身

最短公共超序列



最短公共超序列

■ 递推关系式:

- 令 $len[i, j]$ 表示 $X[1..i]$ 和 $Y[1..j]$ 的最短公共超序列的长度, 则有

$$len[i, j] = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ len[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \min \{ len[i, j-1] + 1, \\ \quad len[i-1, j] + 1 \} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

最短公共超序列

- 示例: $X[1..3] = \text{"abc"}$, $Y[1..3] = \text{"abb"}$

len(i,j)	i=0	1 a	2 b	3 c
j=0	0	1	2	3
1 a	1	1	2	3
2 b	2	2	2	3
3 b	3	3	3	4

伪代码描述

```
1.  for  $i = 0$  to  $n$  do
2.       $len[i, 0] \leftarrow i$ 
3.  for  $j = 0$  to  $m$  do
4.       $len[0, j] \leftarrow j$ 
5.  for  $i = 1$  to  $n$  do
6.      for  $j = 1$  to  $m$  do
7.          if (  $X[i] = Y[j]$  ) then
8.               $len[i, j] \leftarrow len[i - 1, j - 1] + 1$ 
9.          else
10.              $len[i, j] \leftarrow \min\{ len[i - 1, j] + 1, len[i, j - 1] + 1 \}$ 
11. return  $len[n, m]$ 
```

最短公共超序列

- 示例: **X**: a, b, c, d, a, c, e; **Y**: b, a, d, c, a, b, e

len	i=0	1	2	3	4	5	6	7
		a	b	c	d	a	c	e
0	0	1	2	3	4	5	6	7
1 b	1	2 ↑	2 ↖	3 ←	4 ←	5 ←	6 ←	7 ←
2 a	2	2 ↖	3 ↑	4 ↑	5 ↑	5 ↖	6 ←	7 ←
3 d	3	3 ↑	4 ↑	5 ↑	5 ↖	6 ↑	7 ↑	8 ↑
4 c	4	4 ↑	5 ↑	5 ↖	6 ↑	7 ↑	7 ↖	8 ←
5 a	5	5 ↖	6 ↑	6 ↑	7 ↑	7 ↖	8 ↑	9 ↑
6 b	6	6 ↑	6 ↖	7 ↑	8 ↑	8 ↑	9 ↑	10 ↑
7 e	7	7 ↑	7 ↑	8 ↑	9 ↑	9 ↑	10 ↑	10 ↖



最短公共超序列

- 示例: **X**: a, b, c, d, a, c, e; **Y**: b, a, d, c, a, b, e

len	i=0	1	2	3	4	5	6	7
		a	b	c	d	a	c	e
0	0	1	2	3	4	5	6	7
1 b	1	2 ↑	2 ↖	3 ↖	4 ↖	5 ←	6 ←	7 ←
2 a	2	2 ↖	3 ↑	4 ↑	5 ↑	5 ↖	6 ←	7 ←
3 d	3	3 ↑	4 ↑	5 ↑	5 ↖	6 ↑	7 ↑	8 ↑
4 c	4	4 ↑	5 ↑	5 ↖	6 ↑	7 ↑	7 ↖	8 ←
5 a	5	5 ↖	6 ↑	6 ↑	7 ↑	7 ↖	8 ↑	9 ↑
6 b	6	6 ↑	6 ↖	7 ↑	8 ↑	8 ↑	9 ↑	10 ↑
7 e	7	7 ↑	7 ↑	8 ↑	9 ↑	9 ↑	10 ↑	10 ↖

a

b

c

d

a

d

c

a

b

e

伪代码描述

```
1. for  $i = 0$  to  $n$  do
2.    $c[i, 0] \leftarrow i$ 
3. for  $j = 0$  to  $m$  do
4.    $c[0, j] \leftarrow j$ 
5. for  $i = 1$  to  $n$  do
6.   for  $j = 1$  to  $m$  do
7.     if (  $x[i] = y[j]$  ) then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1, b[i, j] \leftarrow 1$ 
8.     else
9.        $c[i, j] \leftarrow \min\{ c[i - 1, j] + 1, c[i, j - 1] + 1 \}$ 
10.      if (  $c[i, j] = c[i - 1, j] + 1$  ) then  $b[i, j] \leftarrow 2$ 
11.      else  $b[i, j] \leftarrow 3$ 
11.  $p \leftarrow n, q \leftarrow m, k \leftarrow 1$  // 回溯构造具体的超序列
12. while (  $p \neq 0$  or  $q \neq 0$  )
13.   if (  $b[p, q] = 1$  ) then {  $SCS[k] \leftarrow y[p], k \leftarrow k + 1, p \leftarrow p - 1, q \leftarrow q - 1$  }
14.   if (  $b[p, q] = 2$  ) then {  $SCS[k] \leftarrow x[p], k \leftarrow k + 1, p \leftarrow p - 1$  }
15.   if (  $b[p, q] = 3$  ) then {  $SCS[k] \leftarrow y[p], k \leftarrow k + 1, q \leftarrow q - 1$  }
```

之后反序输出 $SCS[]$

// 回溯构造具体的超序列

最短公共超序列

■ 定理:

假设 $|X| = m$, $|Y| = n$, $|LCS| = L$, $|SCS| = K$

则有 $L + K = m + n$

■ 示例:

X : a b c d a c e; Y : b a d c a b e

LCS : b d a e; SCS : a b c a d c a c b e

a	b	c		d		a	c		e
	b		a	d	c	a		b	e

a	b	c	a	d	c	a	c	b	e
---	---	---	---	---	---	---	---	---	---

序列对齐与编辑距离

Sequence Alignment and Edit Distance

刘铎

liuduo@bjtu.edu.cn

字符串的相似度

■ 这两个字符串有多相似？

□ occurrence

□ occurrence

o c u r r a n c e -

o c c u r r e n c e

6次错误匹配，1次缺漏

6 mismatches, 1 gap

o c - u r r a n c e

o c c u r r e n c e

1次错误匹配，1次缺漏

1 mismatch, 1 gap

o c - u r r - a n c e

o c c u r r e - n c e

0次错误匹配，3次缺漏

0 mismatches, 3 gaps

编辑距离 (Edit Distance)

■ 应用

- Unix 中 diff 命令的基础
- 语音识别
- 计算生物学

■ 编辑距离 (Edit Distance) [Levenshtein 1966, Needleman-Wunsch 1970]

- 缺漏的惩罚 δ ; 错误匹配的惩罚 α_{pq}
 - 为叙述上的方便, 定义 $\alpha_{pp}=0$
- 总开销 = 缺漏的惩罚总值 + 错误匹配的惩罚总值

C T G A C C T A C C T

C C T G A C T A C A T

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

- C T G A C C T A C C T

C C T G A C - T A C A T

$$2\delta + \alpha_{CA}$$

Levenshtein距离

- Levenshtein 的原始定义：给定两个序列 S_1 和 S_2 ，通过一系列字符编辑（插入、删除、替换）等操作，将 S_1 转变成 S_2
- 完成这种转换所需要的**最少**的编辑操作个数称为 S_1 和 S_2 的**编辑距离**
- 在 Levenshtein 的原始定义中，插入、删除、替换操作中的每一个都具有单位成本，因此 Levenshtein 距离等于字符串转换的最小操作数
- 示例：vintner 转变成 writers，编辑距离 ≤ 5 ：

将v替换为w: vintner
wintner
插入r: wrintner
删除n: wri-tner
删除n: writ-er
插入s: writers

v	-	i	n	t	n	e	r	-
w	r	i	-	t	-	e	r	s

Levenshtein距离

■ 事实上就是:

□ 缺漏的惩罚 $\delta = 1$

□ 错误匹配的惩罚 $\alpha_{pq} = 1$ ($p \neq q$) , $\alpha_{pp} = 0$

将v替换为w: vintner
插入r: **w**rintner
删除n: wri-tner
删除n: writ-er
插入s: writers**s**

v	-	i	n	t	n	e	r	-
w	r	i	-	t	-	e	r	s

LCS距离

- 如 $X: a b c d a c e$; $Y: b a d c a b e$
 - $LCS: b d a e$, LCS距离为 **6** $((7-4)+(7-4))$
 - 实际是SCS长度

a	b	c		d		a	c		e
	b		a	d	c	a		b	e

- 事实上就是:
 - 缺漏的惩罚 $\delta = 1$
 - 错误匹配的惩罚 $\alpha_{pq} = \infty$ ($p \neq q$), $\alpha_{pp} = 0$

序列对齐：解的结构

- 定义 $OPT(i, j)$ 为将字符串 $x_1 x_2 \dots x_i$ 和 $y_1 y_2 \dots y_j$ 对齐的最小总开销
 - 情形1：最优方案选择将 x_i 和 y_j 进行匹配（可能会形成错误匹配）
 - 此时最小总开销为： x_i 和 y_j （可能形成的）错误匹配的惩罚（可能为0）+ 将字符串 $x_1 x_2 \dots x_{i-1}$ 和 $y_1 y_2 \dots y_{j-1}$ 对齐的最小总开销
 - 情形2-1：最优方案选择让 x_i 产生缺漏（无匹配）
 - 此时最小总开销为：一次缺漏惩罚 + 将字符串 $x_1 x_2 \dots x_{i-1}$ 和 $y_1 y_2 \dots y_j$ 对齐的最小总开销
 - 情形2-2：最优方案选择让 y_j 产生缺漏（无匹配）
 - 此时最小总开销为：一次缺漏惩罚 + 将字符串 $x_1 x_2 \dots x_i$ 和 $y_1 y_2 \dots y_{j-1}$ 对齐的最小总开销

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

$$\text{LCS: } \delta = 0; \\ \alpha_{pq} = \infty \ (p \neq q), \ \alpha_{pp} = 0$$

$$\text{SCS: } \delta = 1; \\ \alpha_{pq} = \infty \ (p \neq q), \ \alpha_{pp} = 0$$

序列对齐：算法

Sequence-Alignment ($m, n, x_1x_2...x_m, y_1y_2...y_n, \delta, \alpha$)

```
1. for  $i = 0$  to  $m$  do
```

2. $OPT[0, i] = i \times \delta$

3. **for** $j = 0$ **to** n **do**

4. $OPT[j, 0] = j \times \delta$

5. **for** $i = 1$ **to** m **do**

6. **for** $j = 1$ **to** n **do**

$$7. \quad OPT[i, j] = \min \{ \alpha[x_i, y_j] + OPT[i-1, j-1], \\ \delta + OPT[i-1, j], \\ \delta + OPT[i, j-1] \}$$

```
8. return  $OPT[m, n]$ 
```

■时间复杂度和空间复杂度: $O(mn)$

编辑距离 (Edit Distance)

- 更一般的定义将非负权重函数 $w_{\text{ins}}(x)$ 、 $w_{\text{del}}(x)$ 和 $w_{\text{sub}}(x, y)$ 与操作相关联
- 缺漏的惩罚不再简单地是 $\delta = 1$

图编辑距离 (graph edit distance)

- 图编辑距离 (GED) 用以度量两个图之间的相似性 (或相异性)
- 六种操作
 - 插入一个具有新标号的顶点
 - 删除一个 (通常是孤立的) 顶点
 - 替换顶点, 即更改给定顶点的标号
 - 插入边
 - 删除边
 - 边替换, 即更改给定边的标号

矩阵链乘积

Matrix Chain Multiplication

矩阵链乘积

- 假定给定了矩阵的序列 A_1, A_2, \dots, A_n
 - 其中 A_i 为 $P_{i-1} \times P_i$ 阶矩阵
 - 于是 A_{i-1} 和 A_i 都是可以进行乘法的
- 目的是计算它们的链乘积 $A_1 A_2 \dots A_n$
 - 然而每次只能是两个矩阵相乘得到第三个矩阵
- 由于矩阵乘法满足结合律，因此无论计算的过程是什么样的，其最终结果都是一致的

矩阵链乘积

■ 例：有4个矩阵 A_1, A_2, A_3, A_4

□ 将要计算乘积 $A_1A_2A_3A_4$

■ 有5种可能的计算过程

1. $(A_1(A_2(A_3A_4)))$

2. $(A_1((A_2A_3)A_4))$

3. $((A_1A_2)(A_3A_4))$

4. $((A_1(A_2A_3))A_4)$

5. $(((A_1A_2) A_3) A_4)$

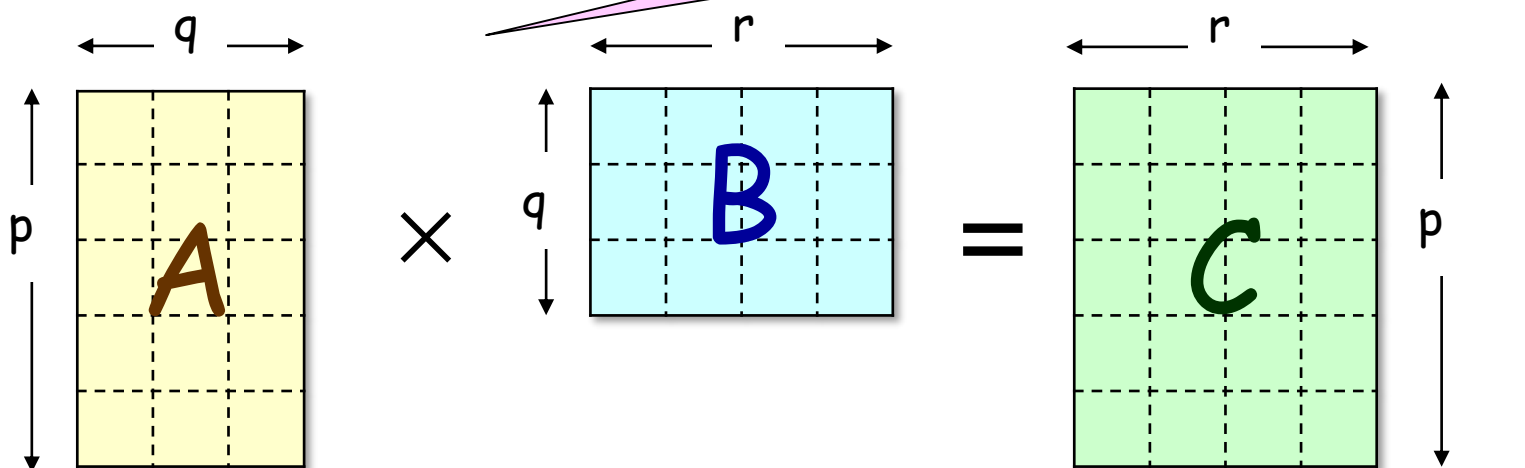
矩阵链乘积

- 虽然无论计算的过程是什么样的，其最终结果都是一致的
- 然而，不同的计算过程的效率可能是不同的

矩阵链乘积

■ 先来看两个矩阵相乘的复杂度

□ 采用“教科书算法”



$$c_{ij} = \sum_{k=1}^q a_{ik} b_{kj}$$

$k=1$

矩阵链乘积

输入：矩阵 $A_{p \times q}$ 和 $B_{q \times r}$ （维数分别是 $p \times q$ 和 $q \times r$ ）

输出：矩阵 $C_{p \times r} = A \cdot B$

MATRIX-MULTIPLY ($A_{p \times q}, B_{q \times r}$)

1. **for** $i \leftarrow 1$ **to** p
2. **for** $j \leftarrow 1$ **to** r
3. $C(i, j) \leftarrow 0$
4. **for** $k \leftarrow 1$ **to** q
5. $C(i, j) \leftarrow C(i, j) + A(i, k) \cdot B(k, j)$
6. **return** C

总的元素乘法次数
 pqr



矩阵链乘积

■ 示例：考虑3个矩阵 $A_{10 \times 100}$, $B_{100 \times 5}$ 和 $C_{5 \times 50}$

■ 有两种“加括号”的方法：

□ $((AB)C) = D_{10 \times 5} \cdot C_{5 \times 50}$

- $AB \Rightarrow 10 \cdot 100 \cdot 5 = 5,000$ 次元素乘法
 - $DC \Rightarrow 10 \cdot 5 \cdot 50 = 2,500$ 次元素乘法
- } 总计：7,500

□ $(A(BC)) = A_{10 \times 100} \cdot E_{100 \times 50}$

- $BC \Rightarrow 100 \cdot 5 \cdot 50 = 25,000$ 次元素乘法
 - $AE \Rightarrow 10 \cdot 100 \cdot 50 = 50,000$ 次元素乘法
- } 总计：75,000

总计：75,000

矩阵链乘积 —— 问题定义

■ 下面来定义矩阵链乘积问题

- 给定了矩阵的序列 A_1, A_2, \dots, A_n ，其中 A_i 的阶数为 $P_{i-1} \times P_i$
- 试确定矩阵相乘的次序（即，加括号的方法）使得矩阵元素相乘的总次数最少


■ 注意：不是真的计算该乘积

矩阵链乘积 —— 蛮力法

■ 蛮力法？

□ 尝试所有可能的次序？

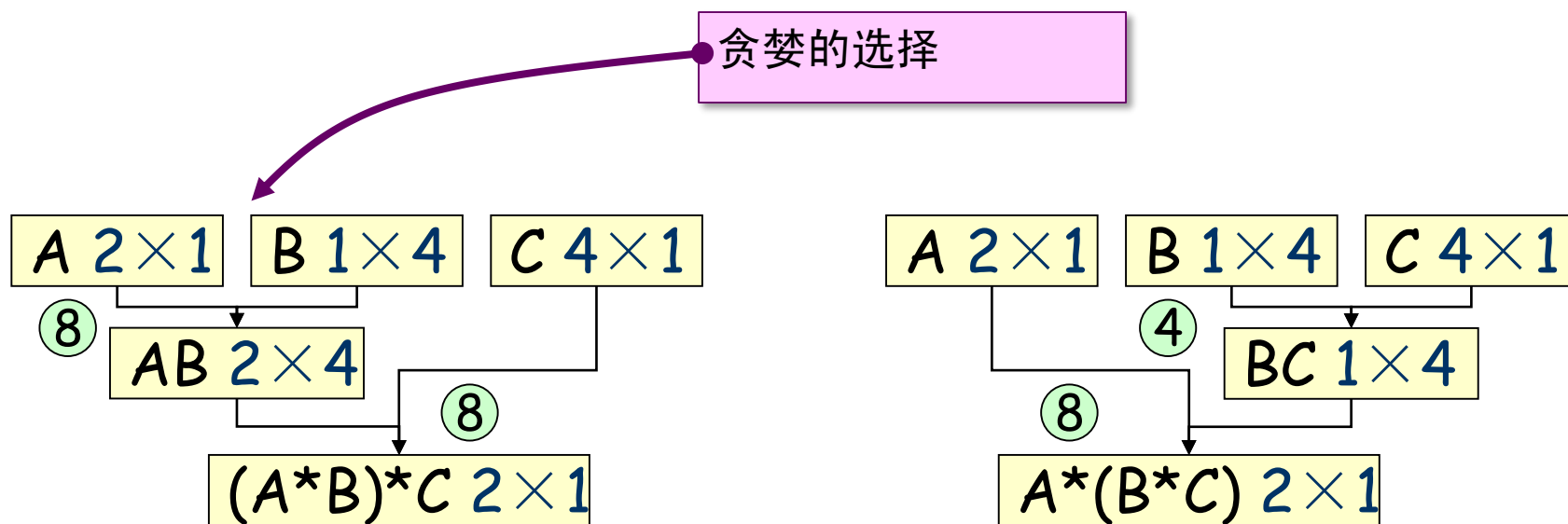
卡特兰数
Catalan number


$$C(n-1) = \frac{1}{n} \binom{2n-2}{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

粗略地讲，时间为 $O(4^n)$

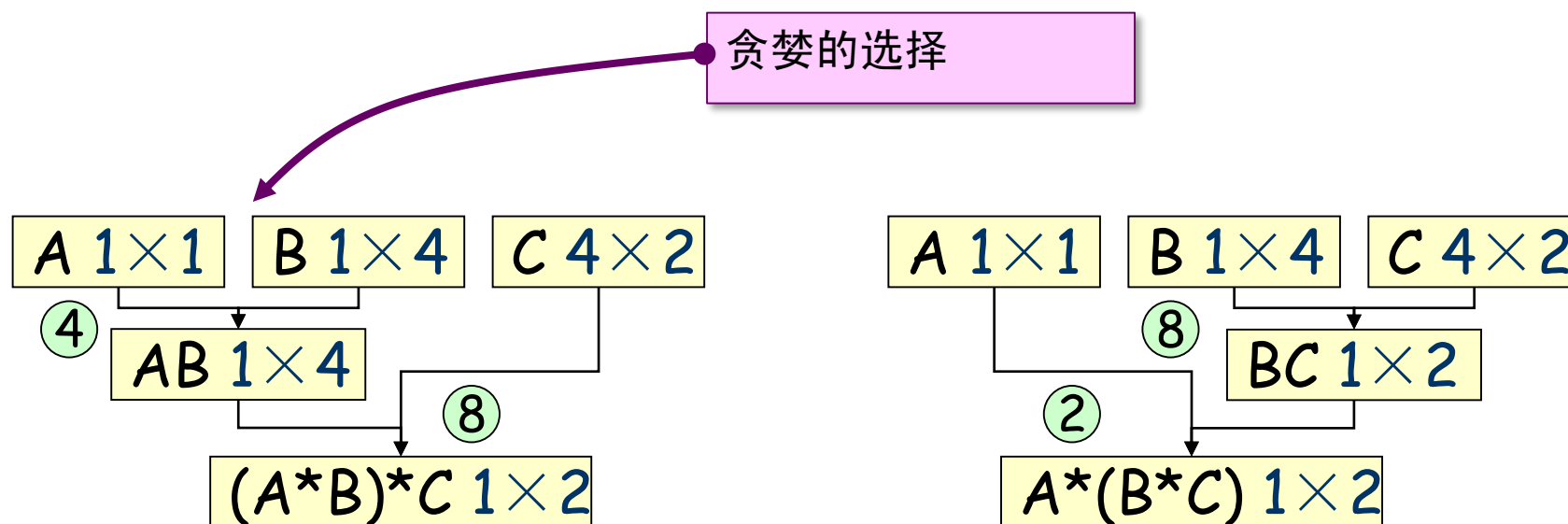
矩阵链乘积 —— 贪婪策略？

- 设想1：每次都选择使用最多元素乘法次数的两个矩阵相乘
- 然而，反例：



矩阵链乘积 —— 贪婪策略？

- 设想2：每次都选择使用最少元素乘法次数的两个矩阵相乘
- 然而，反例：



矩阵链乘积 —— 递推式

■ 考察最后一次乘法

□ 它将矩阵序列划分为两部分

■ 例

1. $(A_1 (A_2 (A_3 A_4)))$

2. $(A_1 ((A_2 A_3) A_4))$

3. $((A_1 A_2) (A_3 A_4))$

4. $((A_1 (A_2 A_3)) A_4)$

5. $(((A_1 A_2) A_3) A_4)$

矩阵链乘积 —— 递推式

- 将 $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ 的乘积记做 $A_{i..j}$ ($j \geq i$)
 - 其中共有 $l = j - i + 1$ 个矩阵
- 令 $m(i, j)$ 表示计算 $A_{i..j}$ 的最优方式的元素乘法总次数
- 当 $j = i$ 时, $m(i, j) = 0$

矩阵链乘积 —— 递推式

- 假设计算 $A_{i..j}$ ($j>i$) 的最后一次矩阵乘法是

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

其中 $i \leq k < j$

- $A_{i..k}$ 的阶数为 $P_{i-1} \times P_k$, $A_{k+1..j}$ 的阶数为 $P_k \times P_j$
- 这次矩阵乘法的开销是 $P_{i-1} \times P_k \times P_j$

矩阵链乘积 —— 递推式

- 假设计算 $A_{i..j}$ ($j > i$) 的最后一次矩阵乘法是

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

其中 $i \leq k < j$

- 如果在固定 k 的前提下希望将总的元素乘法总次数降到最少，那么之前就应该按照最优方式计算 $A_{i..k}$ 和 $A_{k+1..j}$
 - （请思考其原因）

矩阵链乘积 —— 递推式

- 假设固定计算 $A_{i..j}$ ($j>i$) 的最后一次矩阵乘法为

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

- 这次矩阵乘法的开销是 $P_{i-1} \times P_k \times P_j$
- 之前计算 $A_{i..k}$ 和 $A_{k+1..j}$ 的最优方式的开销分别是 $m(i, k)$ 和 $m(k+1, j)$
- 于是总开销的最小可能就是

$$m(i, k) + m(k+1, j) + P_{i-1} \times P_k \times P_j$$

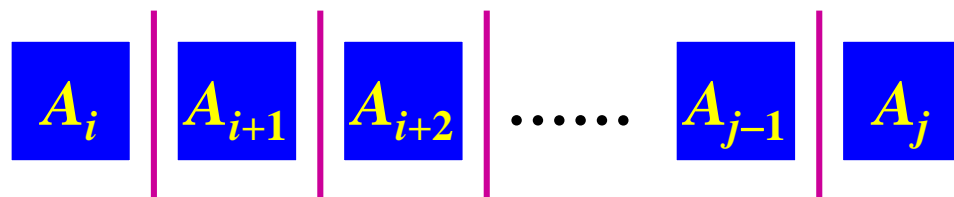
矩阵链乘积 —— 递推式

- 但是...

假设**固定**计算 $A_{i..j}$ 的最后一次矩阵乘法为

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

- 如何知道计算 $A_{i..j}$ 的最优方案的最后一次矩阵乘法**发生在哪里**？即， $k = ?$
- 无论如何，最后一次乘法**必然会发生**在某个 A_k “后面”
- 于是就把所有的 $i \leq k < j$ **都**试试看，然后选择其中“**最少**”的



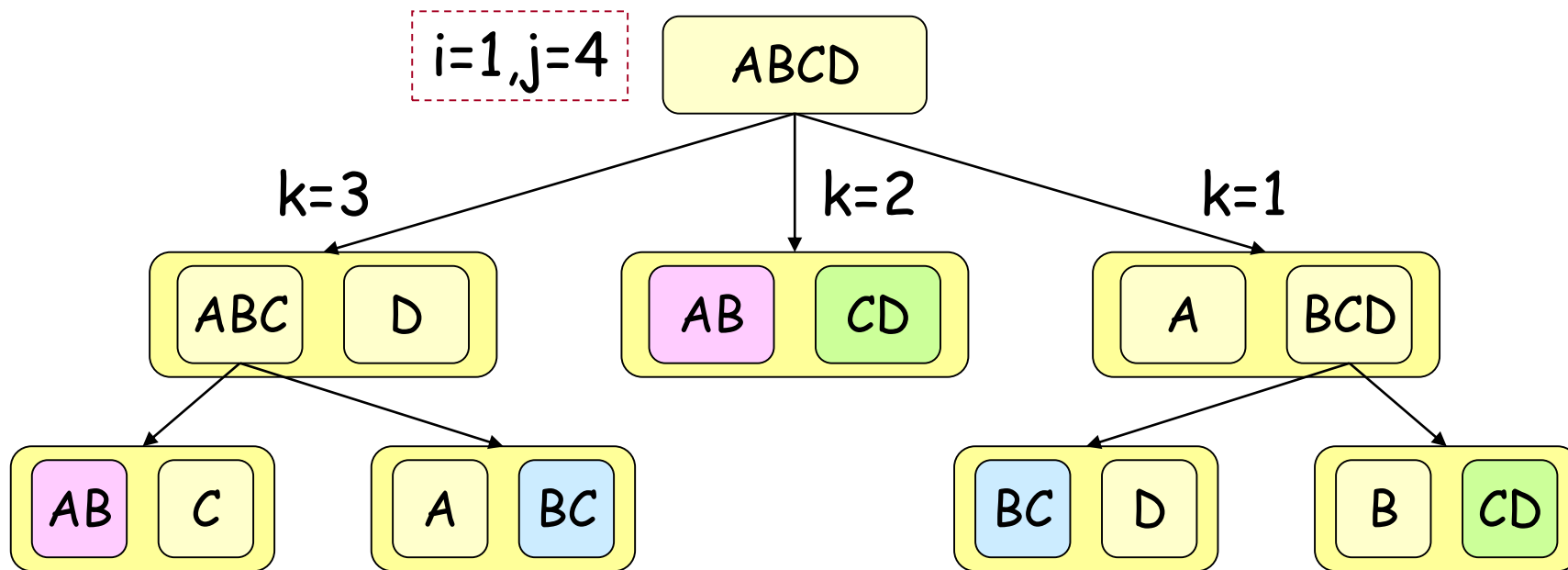
矩阵链乘积 —— 递推式

■ 于是得到递推式

$$m(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m(i, k) + m(k+1, j) + P_{i-1}P_kP_j \} & \text{if } i < j \end{cases}$$

矩阵链乘积 —— 递归方式

■ 如果采用递归方式



重复工作!!!

时间复杂度 $\Omega(2^{n-1})$

矩阵链乘积—— 动态规划

$$m(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m(i, k) + m(k+1, j) + P_{i-1}P_kP_j \} & \text{if } i < j \end{cases}$$

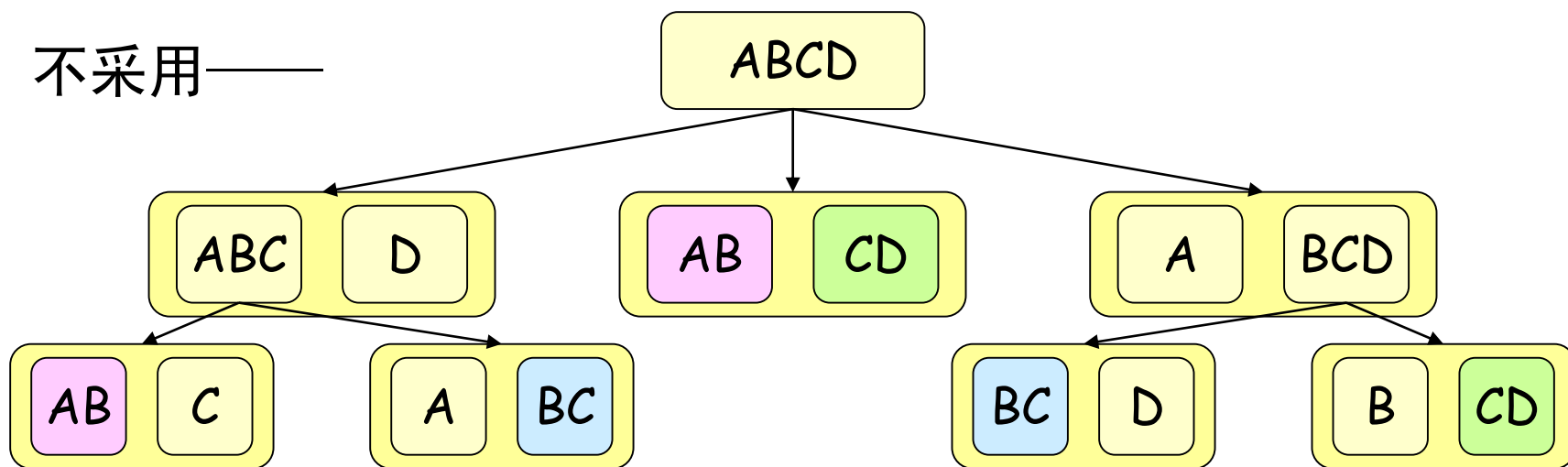
i \ j	1	2	3	4	5	6	7	8	9	10
1	0									
2		0								
3			0							
4				0						
5					0					
6						0				
7							0			
8								0		
9									0	
10										0

Diagram illustrating the dynamic programming table for matrix chain multiplication. The table shows the minimum number of scalar multiplications required to compute the product of matrices from index i to j . The diagonal elements are 0. The table is partitioned into regions by a large pink arrow and a large grey arrow. The pink arrow points from the bottom-left to the top-right, representing the sequence of subproblems. The grey arrow points from the top-left to the bottom-right, representing the sequence of subproblems. The label $(i, k < j)$ is placed in the upper-left region, and the label $(k+1 > i, j)$ is placed in the lower-right region.

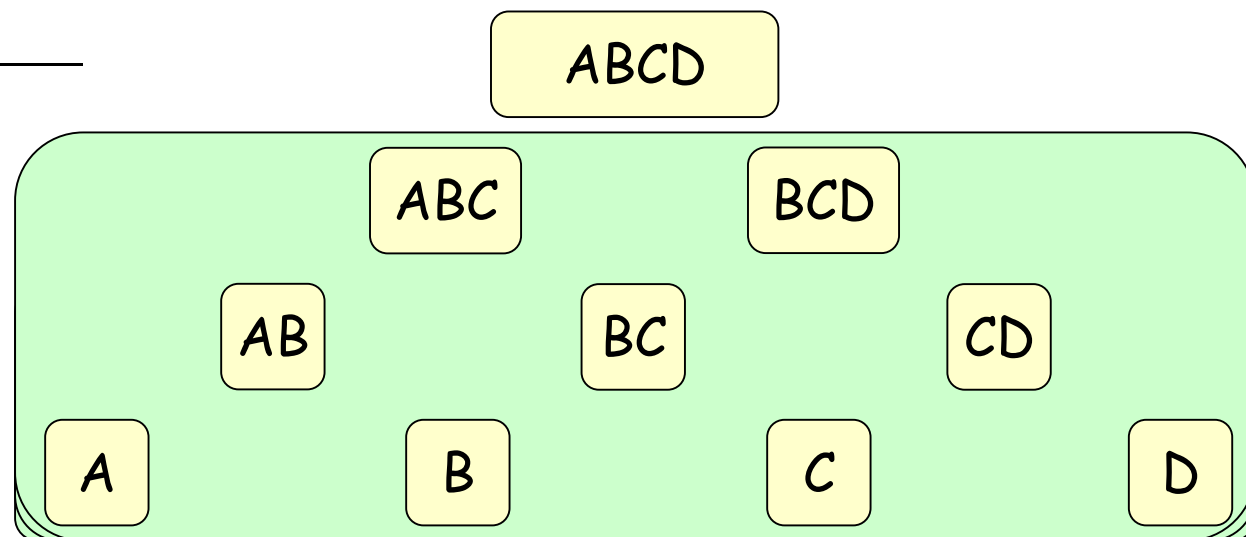
.....
$j - i + 1 = 5$
$j - i + 1 = 4$
$j - i + 1 = 3$
$j - i + 1 = 2$
$j - i + 1 = 1$

矩阵链乘积—— 动态规划

不采用——



而是——



矩阵链乘积—— 动态规划

- 首先计算长度 $l = 1$ 的链乘积的最优开销
- 然后，对于长度 $l = 2, 3, \dots$ 的链乘积，自下而上地计算其最优开销

矩阵链乘积—— 动态规划

■ 填最优开销表 m 和划分表 s

□ $s(i, j) = 0$

□ $s(i, j)$ ($j > i$) 表示计算 $A_{i..j}$ 的最优方法的最后一次矩阵乘法发生的位置，即之前提及的 k 值

MATRIX-CHAIN-ORDER

输入：序列 $P_0, P_1, P_2, \dots, P_n$

输出：最优开销表 m 和划分表 s

1. **for** $i = 1$ **to** n
2. $m(i, i) \leftarrow 0, s(i, i) \leftarrow 0$
3. **for** $l = 2$ **to** n
4. **for** $i = 1$ **to** $n - l + 1$
5. $j \leftarrow i + l - 1$
6. $m(i, j) \leftarrow \infty$
7. **for** $k = i$ **to** $j - 1$
8. $q \leftarrow m(i, k) + m(k+1, j) + P(i-1) \cdot P(k) \cdot P(j)$
9. **if** $q < m(i, j)$ **then**
10. $m(i, j) \leftarrow q, s(i, j) \leftarrow k$
11. **return** m 和 s

时间复杂度： $O(n^3)$

空间复杂度： $O(n^2)$

矩阵链乘积 —— 示例

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

■ 最小开销 15,125

■ 最优方案

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

如何得到的？

矩阵链乘积 —— 示例

■ 计算的次序:

□ $m(1, 1), m(2, 2), m(3, 3), \dots, m(6, 6)$

// 设置关于 A_i 的初值

□ $m(1, 2), m(2, 3), m(3, 4), m(4, 5), m(5, 6)$

// $l=2$ $A_i \cdot A_{i+1}$

□ $m(1, 3), m(2, 4), m(3, 5), m(4, 6),$

// $l=3$ $A_i \cdot \dots \cdot A_{i+2}$

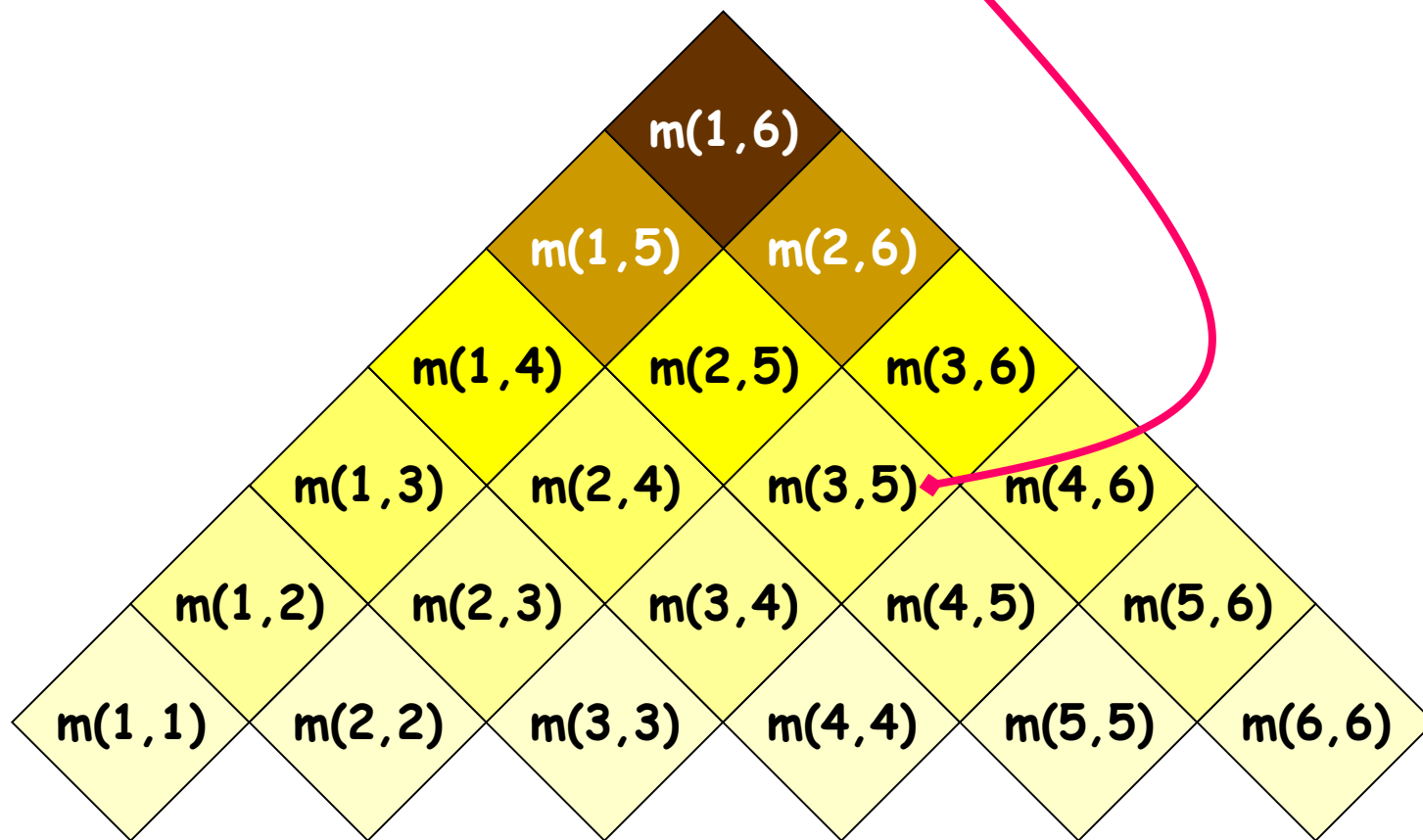
□

□ $m(1, 6)$

// $l=6$ $A_1 \cdot A_2 \cdot \dots \cdot A_6$

示例

$$m[3, 5] = \min \{m[3, k] + m[k+1, 5] + P_2 P_k P_5\} \\ 3 \leq k < 5$$



示例

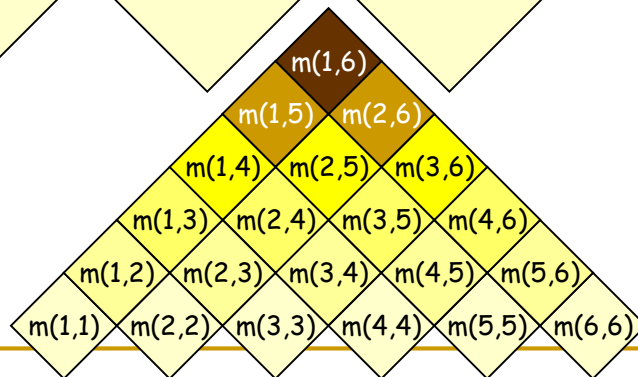
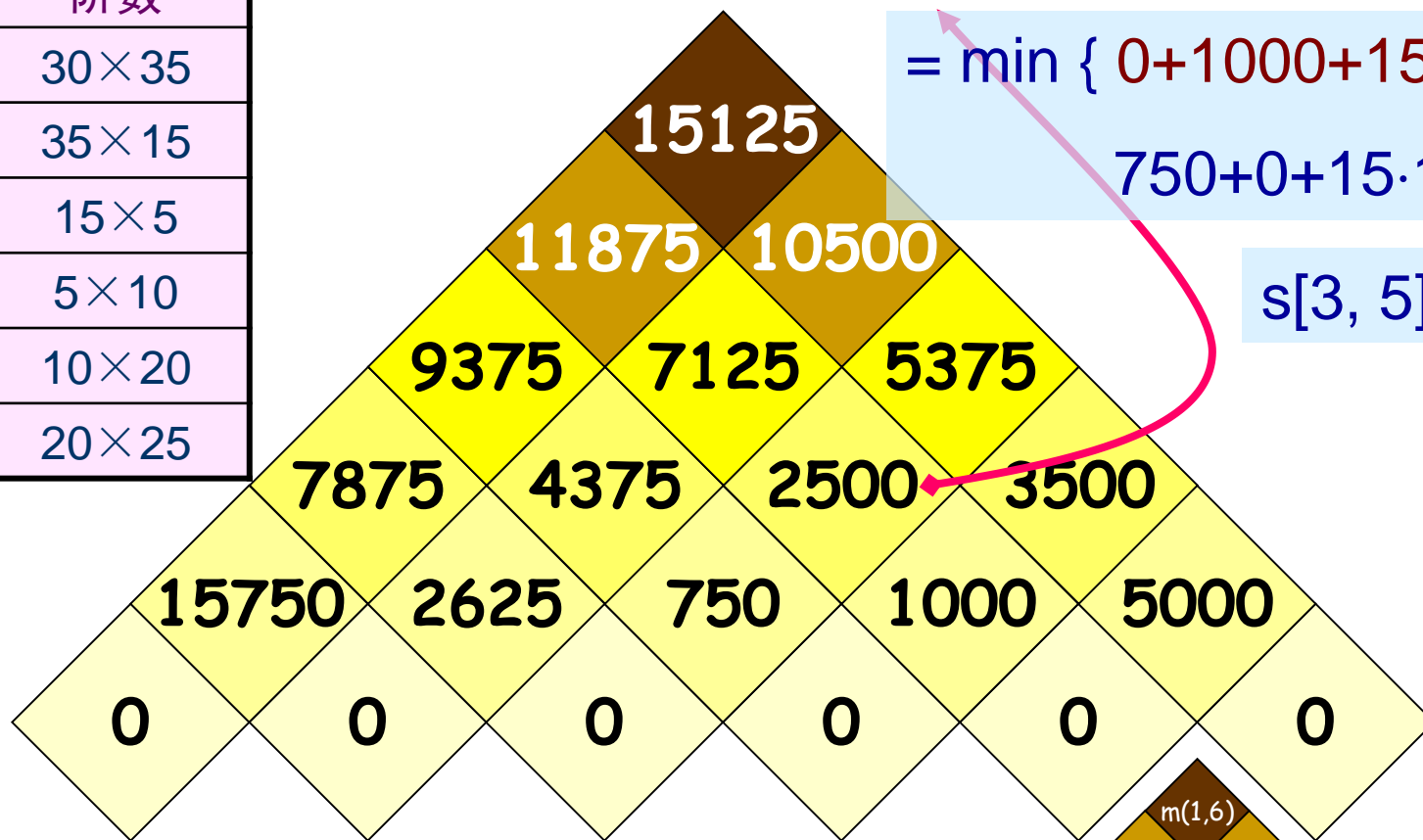
矩阵	阶数
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$$m[3, 5] = \min \{ m[3, 3] + m[4, 5] + P_2 P_3 P_5,$$

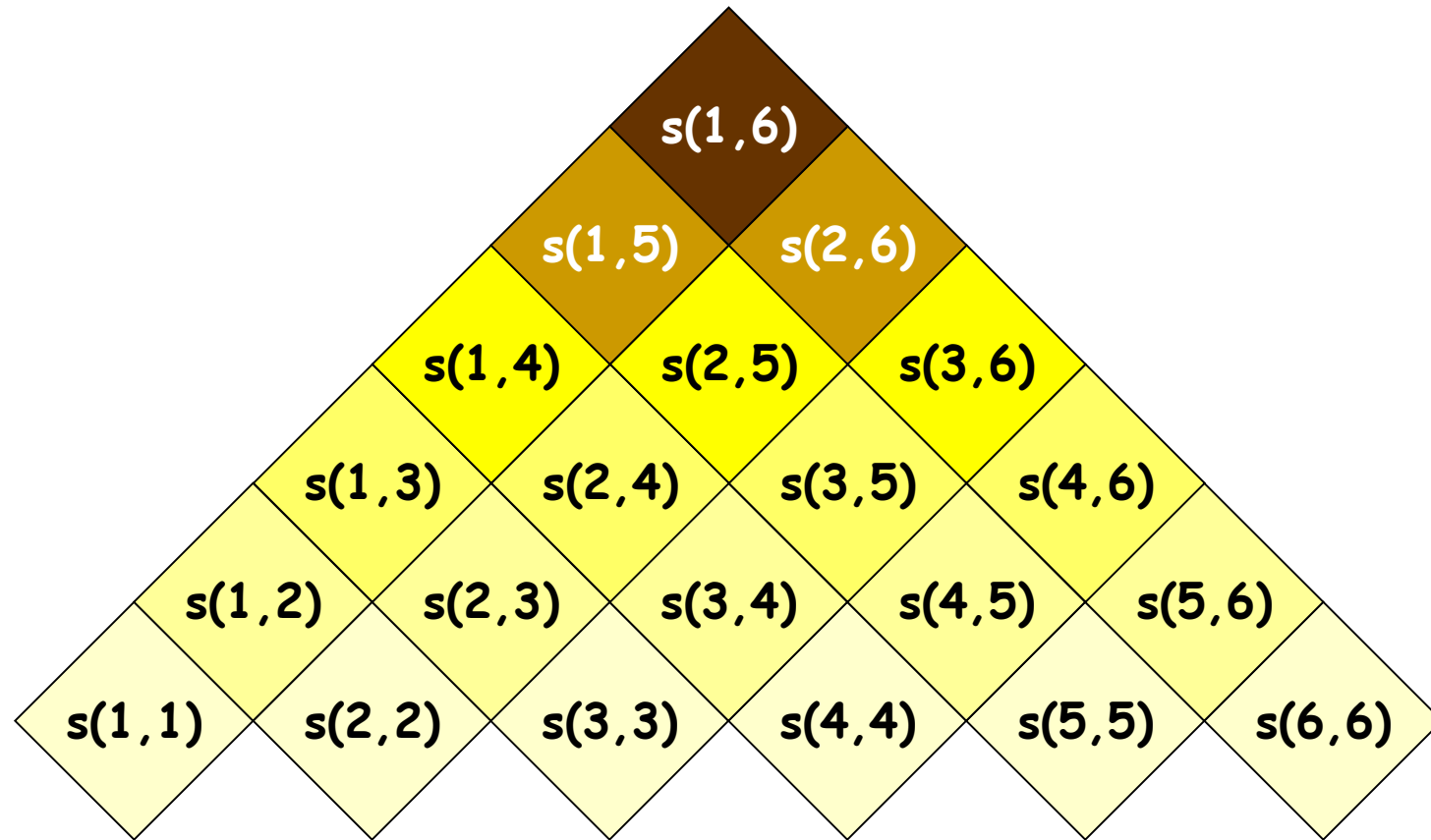
$$m[3, 4] + m[5, 5] + P_2 P_4 P_5 \}$$

$$= \min \{ 0 + 1000 + 15 \cdot 5 \cdot 20, \\ 750 + 0 + 15 \cdot 10 \cdot 20 \}$$

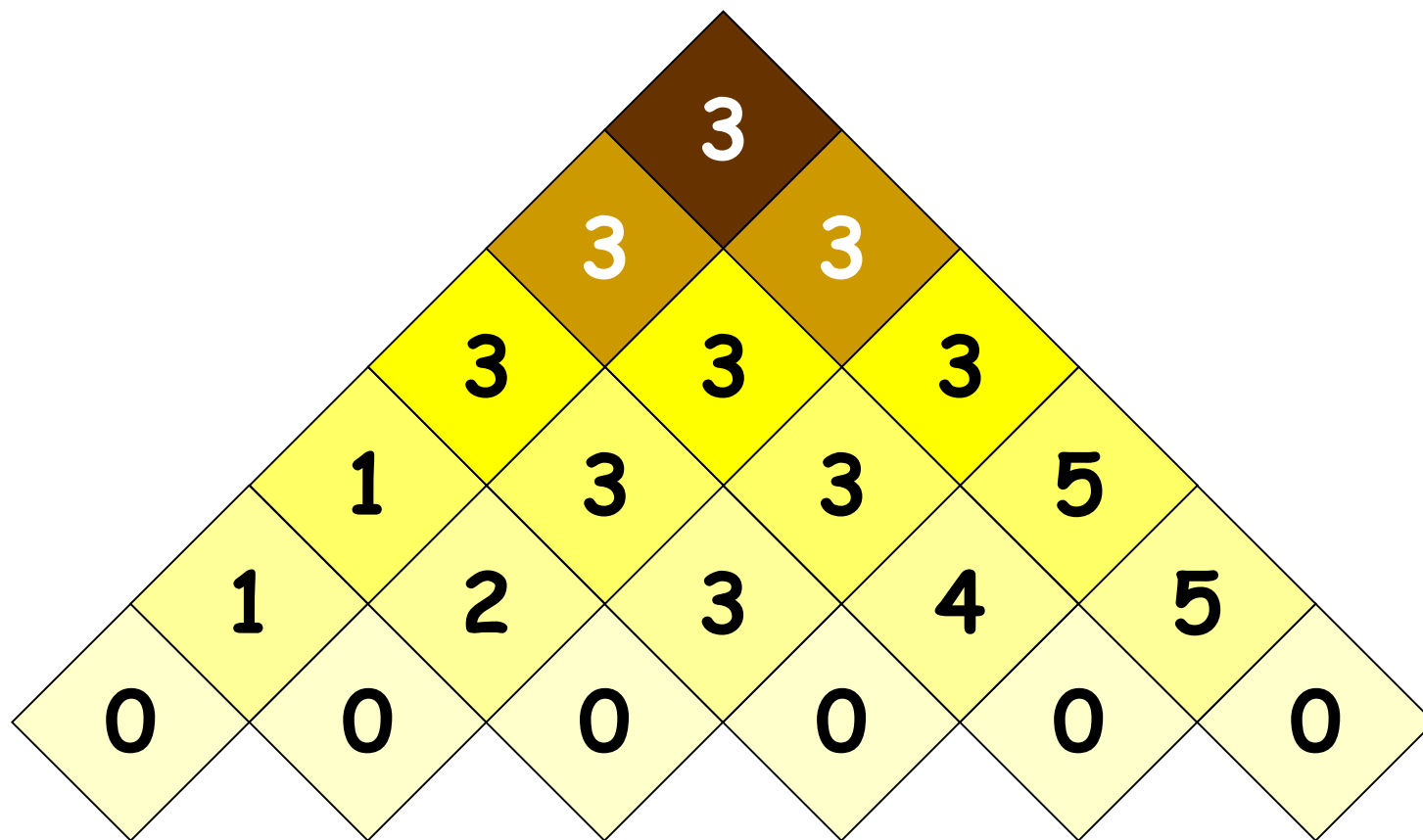
$$s[3, 5] = 3$$



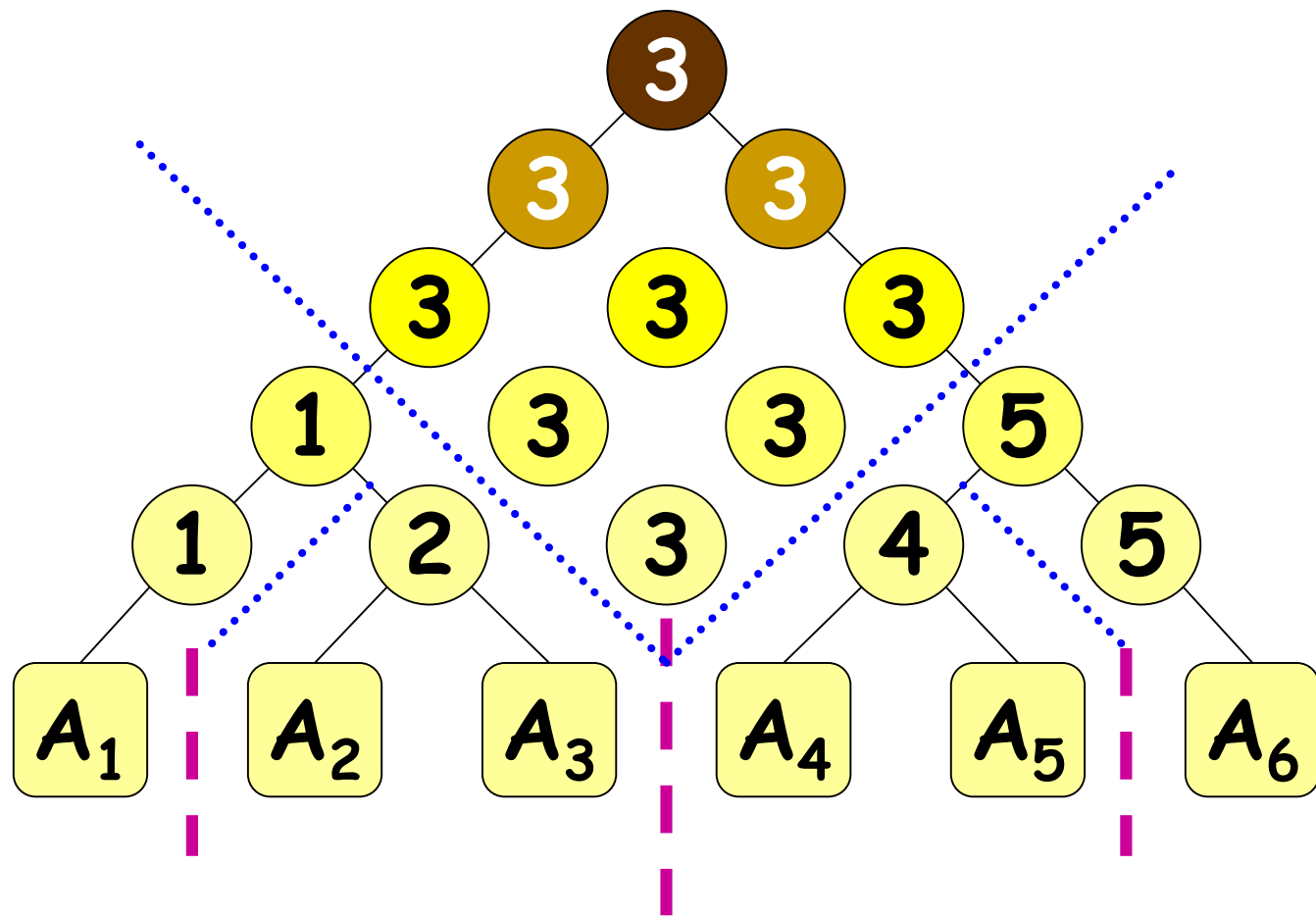
示例



示例

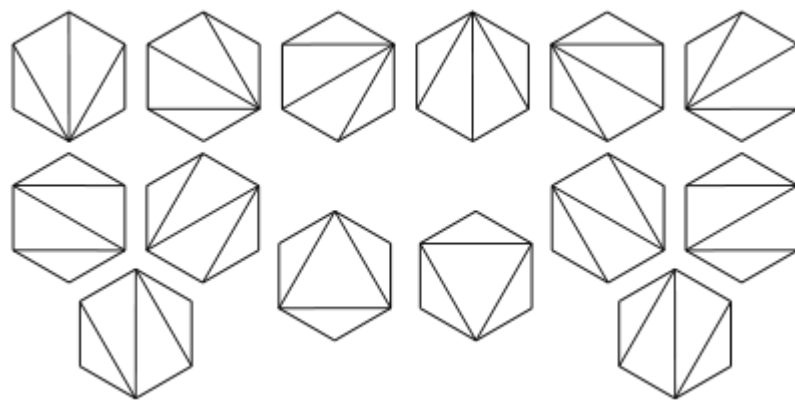


示例



更为有效的算法

- 1984年，Hu and Shing 发表了一个 $n \cdot \log n$ 复杂度的算法
- 基本思想：将矩阵链乘积问题转化（或称归约）为将凸多边形划分为不相交三角形的问题



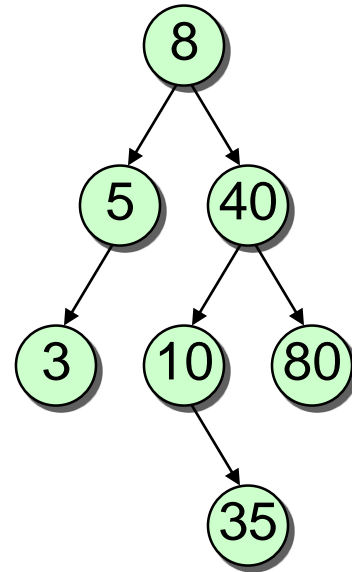
- 而后，他们设计了一个 $n \cdot \log n$ 时间的划分算法

最优二叉查找树

Optimal BST

二叉查找树

- 如果二叉树的任一顶点的键值都大于其非空左子树的所有顶点的键值，而小于其非空右子树的所有顶点的键值，则称其为一棵**二叉查找树**（Binary Search Tree, BST）
- 对一棵二叉查找树进行中序遍历，所得的键值序列一定是递增有序的



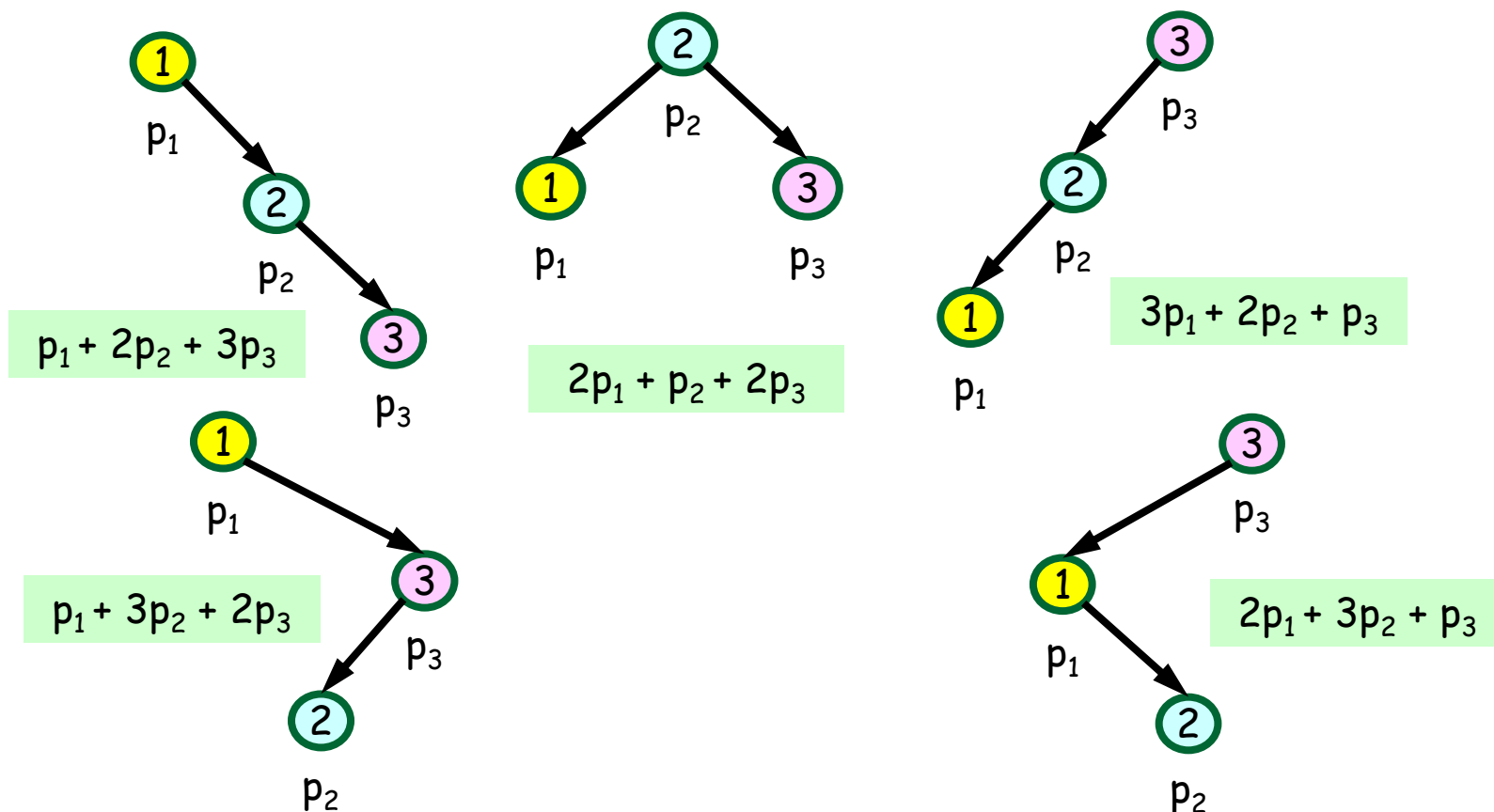
最优二叉查找树 —— 问题描述

■ 问题描述

- 给定 n 个不同的键值 k_1, k_2, \dots, k_n , 每个键值 k_i 被访问的概率为 p_i
- 假设 $k_1 < k_2 < \dots < k_n$
- $p_1 + p_2 + \dots + p_n = 1$
- 应如何构建二叉查找树以最小化其成功查找的期望开销（比较次数）？

最优二叉查找树 —— 问题描述

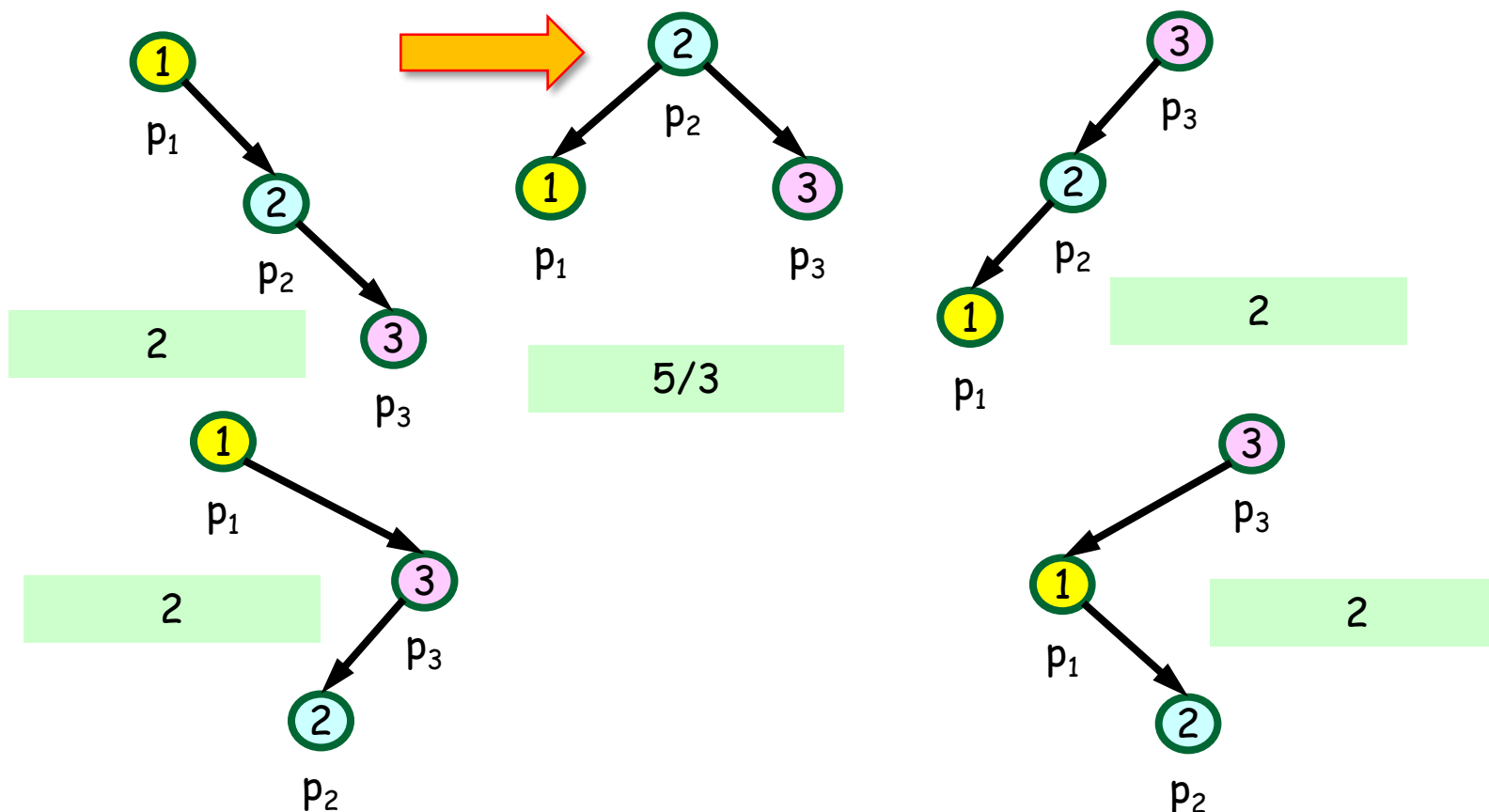
- 例1：键值为1、2和3，查找概率分别为 p_1 、 p_2 和 p_3 。所有可能的5棵二叉查找树及其成功查找的期望开销（比较次数）为



最优二叉查找树 —— 问题描述

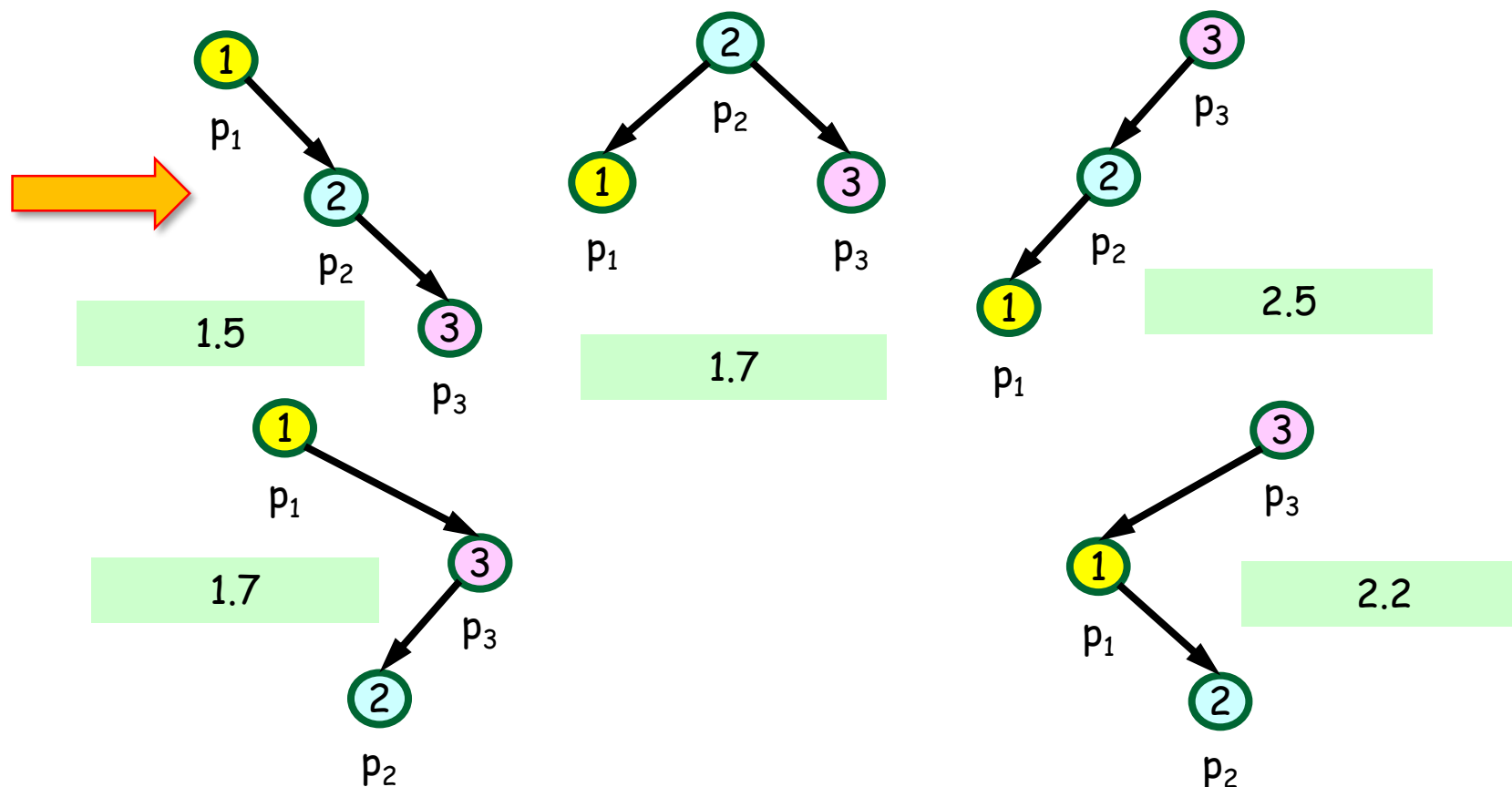
■ 当 $p_1=p_2=p_3=1/3$ 时，第二棵树是最优的

□ 事实上，平衡树就是所有键值的查找概率都相等时的特例



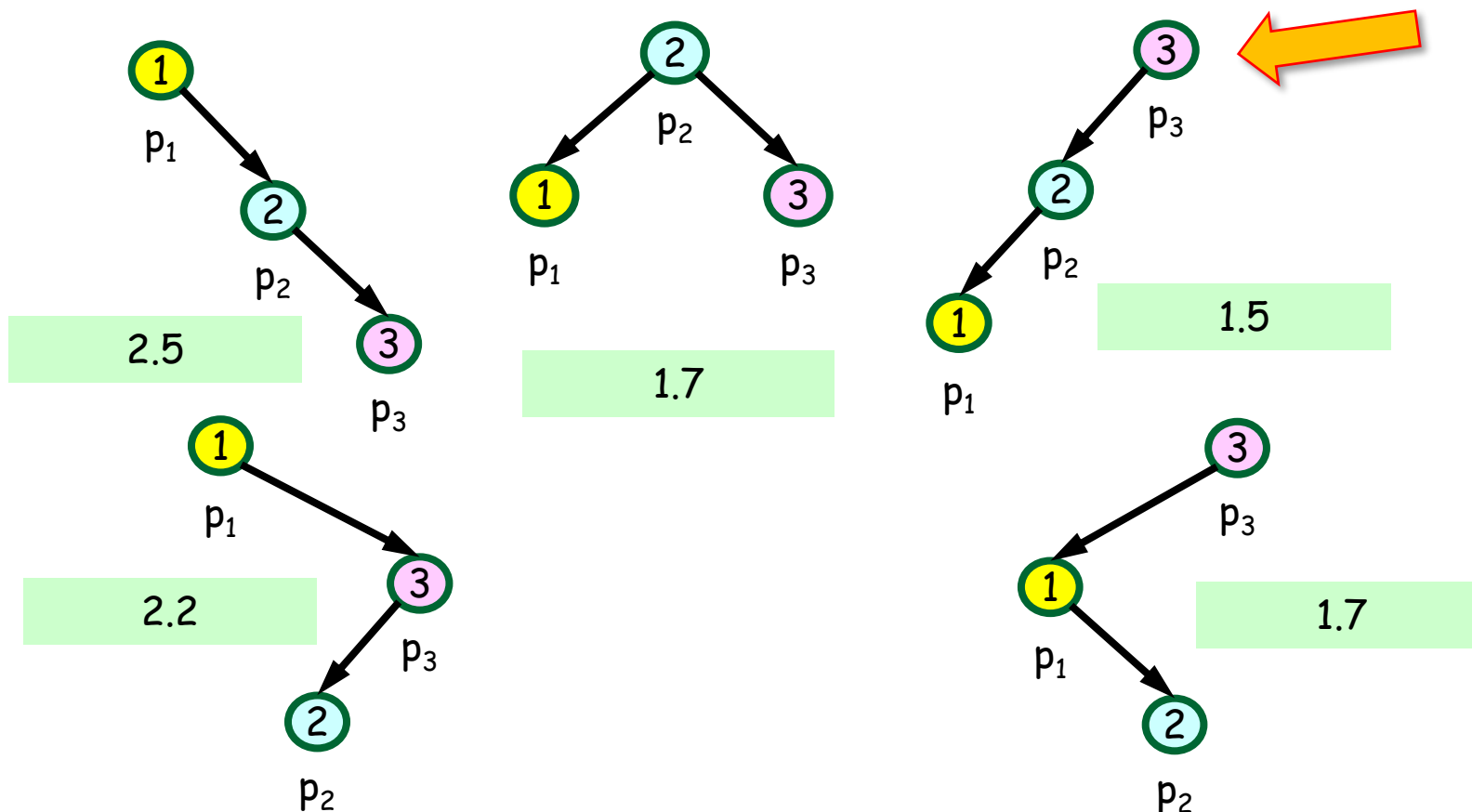
最优二叉查找树 —— 问题描述

- 当 $p_1=0.6$ 、 $p_2=0.3$ 、 $p_3=0.1$ 时，第一棵树是最优的



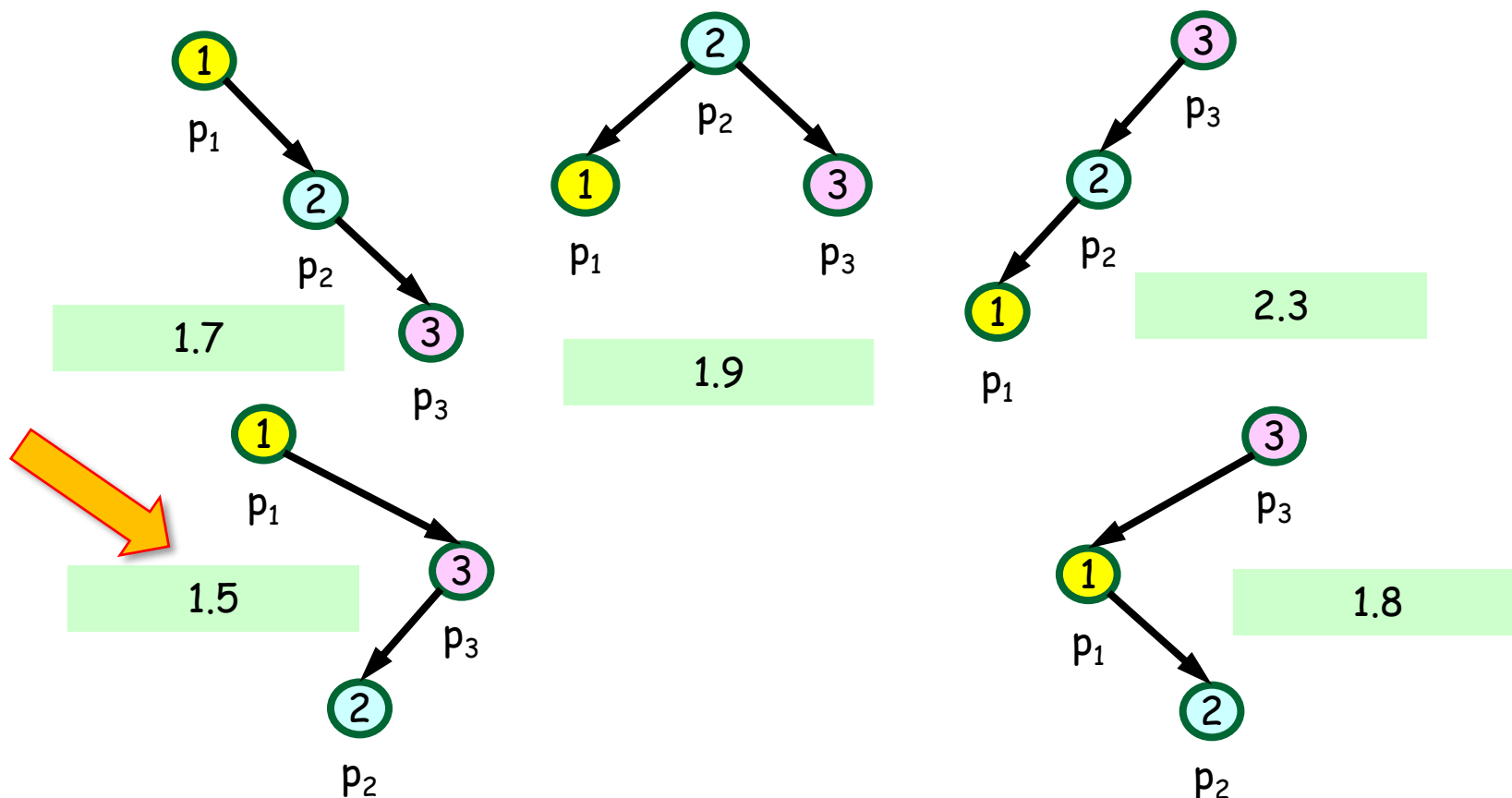
最优二叉查找树 —— 问题描述

- 当 $p_1=0.1$ 、 $p_2=0.3$ 、 $p_3=0.6$ 时，第三棵树是最优的



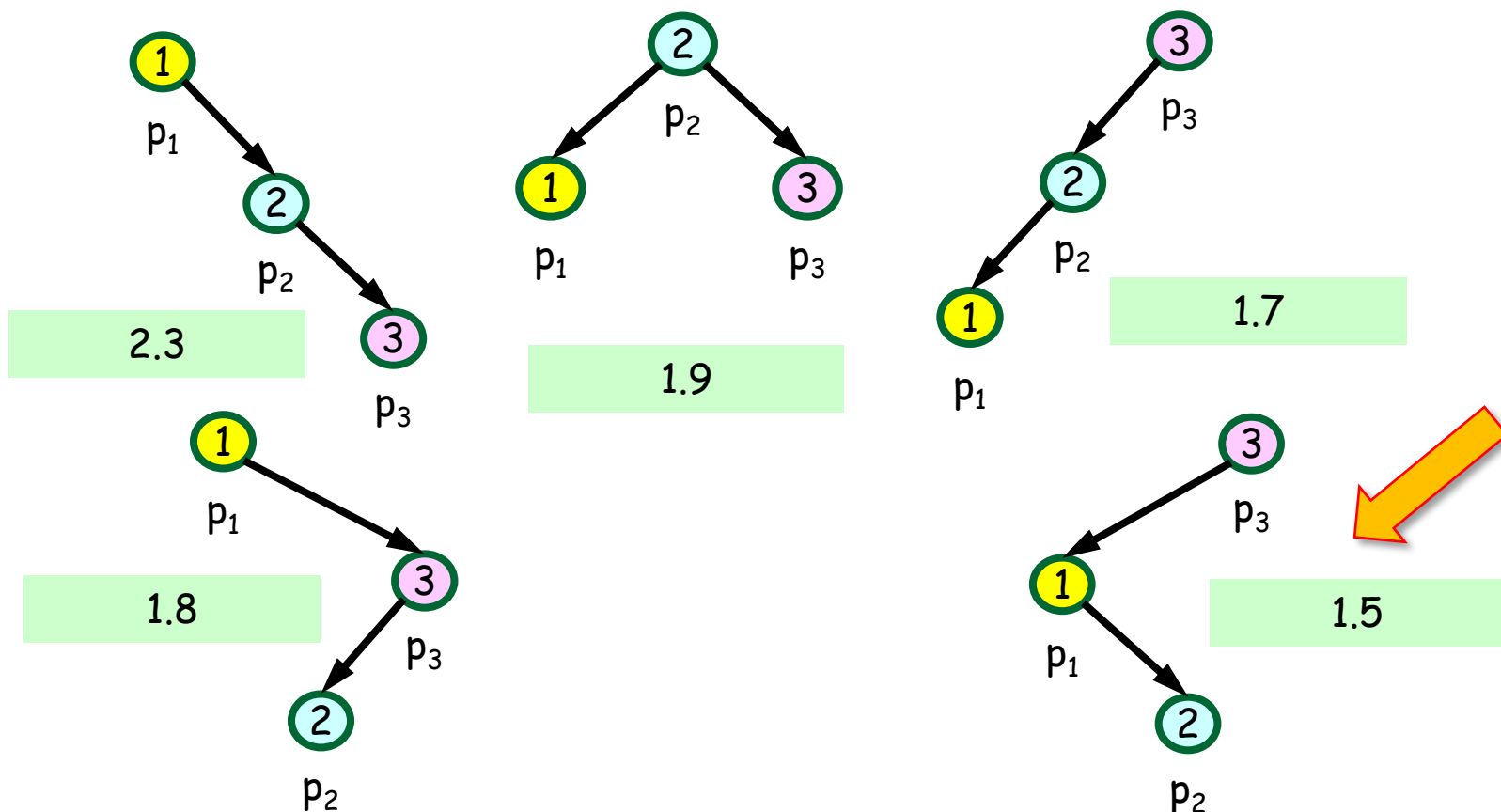
最优二叉查找树 —— 问题描述

- 当 $p_1=0.6$ 、 $p_2=0.1$ 、 $p_3=0.3$ 时，第四棵树是最优的



最优二叉查找树 —— 问题描述

- 当 $p_1=0.3$ 、 $p_2=0.1$ 、 $p_3=0.6$ 时，第五棵树是最优的



最优二叉查找树 —— 问题描述

- 所以，我们不是要试图构建一棵平衡树，而是要优化道路的加权长度
- 而且，此时内部节点也包含键值，并因“查找”而固定了树中节点的中序次序，因此和Huffman编码的问题背景和适用条件不同
- 基本原则：具有更大访问概率的键值的顶点应更接近于根

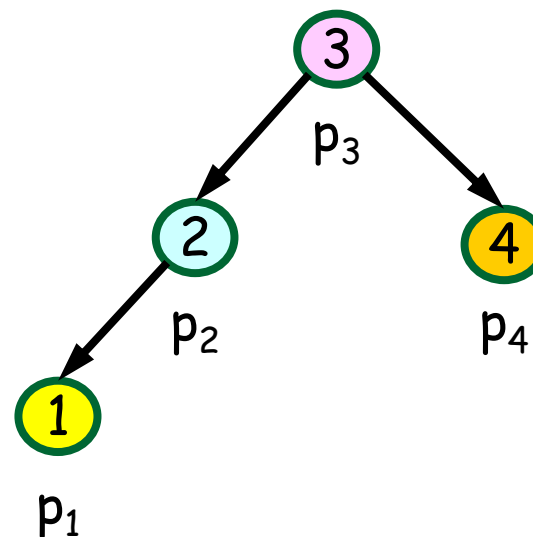
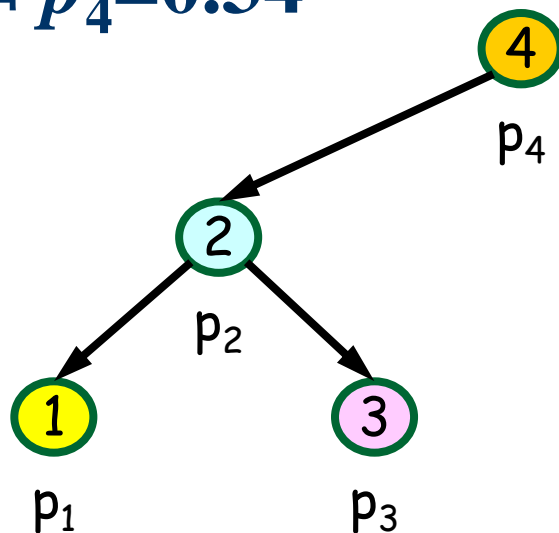
最优二叉查找树 —— 蛮力法？

- 所有可能的二叉查找树共有 Catalan数 C_n 棵，因此蛮力法逐一验证效率极低

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

最优二叉查找树 —— 贪婪策略？

- 将查找概率最大的键值作为根？
- 反例：键值为1、2、3和4，查找概率分别为 $p_1=0.22$ 、 $p_2=0.22$ 、 $p_3=0.22$ 和 $p_4=0.34$



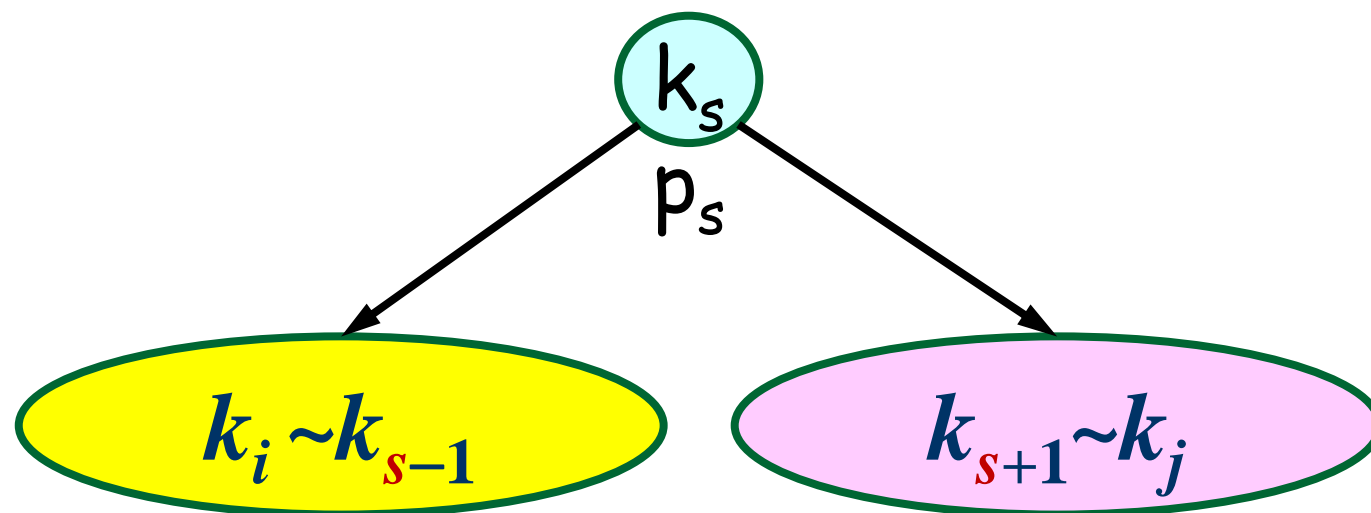
- 左侧树（查找概率最大的键值作为根）成功查找的期望开销为**2.1**，而右侧树成功查找的期望开销为**2**

最优二叉查找树 —— 递推关系

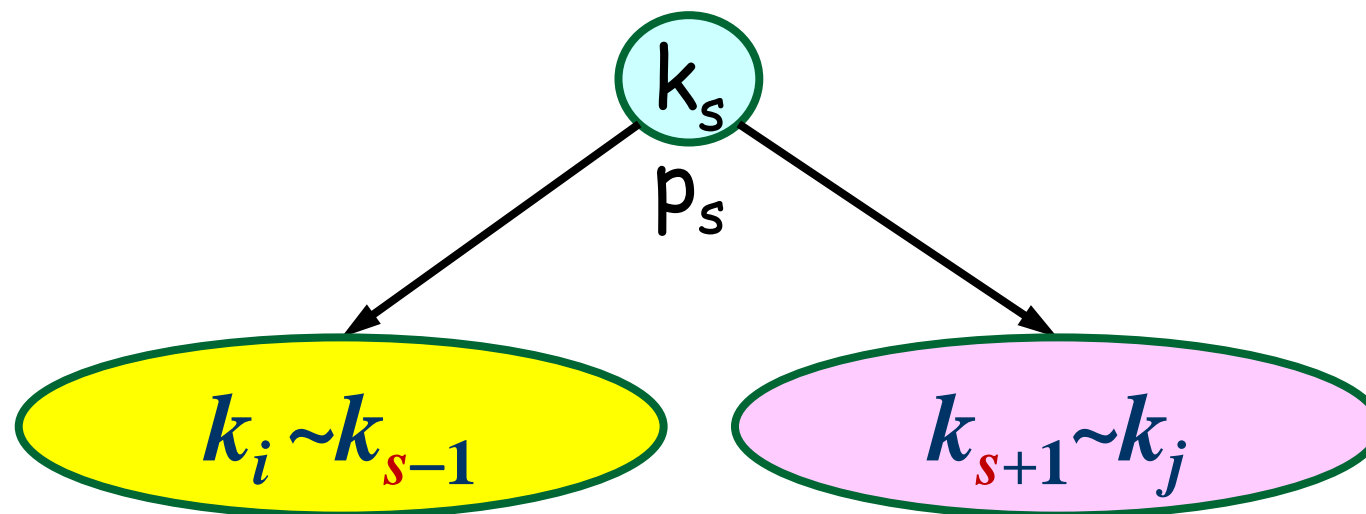
- 记 $T(i, j)$ 表示对应于键值 k_i, \dots, k_j （及其相应的查找概率）的最优二叉查找树
 - 此时不要求 $p_i + \dots + p_j = 1$
- 并令 $C(i, j)$ 表示此时成功查找的（最优）期望开销（比较次数）
- $i > j$ 时， $T(i, j)$ 为空树， $C(i, j) = 0$

最优二叉查找树 —— 递推关系

- 考虑 $T(i, j)$ 的根
- 如果根的键值是 k_s ($i \leq s \leq j$)，那么其左子树键值为 $k_i \sim k_{s-1}$ ，右子树键值为 $k_{s+1} \sim k_j$



最优二叉查找树 —— 递推关系



■ 此时

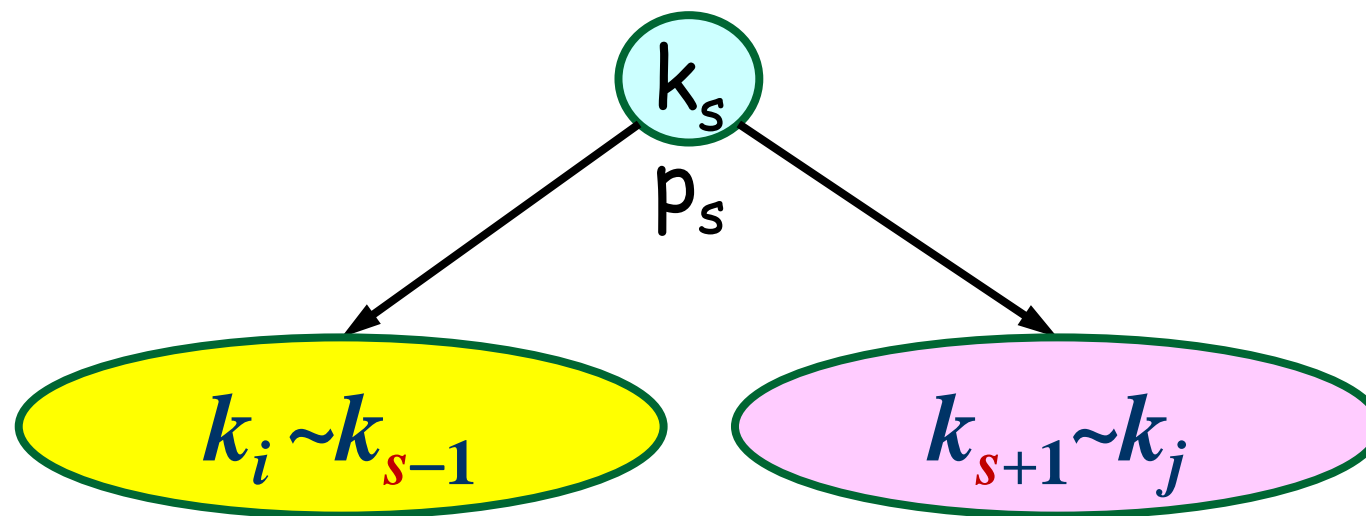
■ $C(i, j) = p_s \times 1 +$

$\text{cost}_{\text{left}} + (p_i + \dots + p_{s-1}) \times 1 +$
 $\text{cost}_{\text{right}} + (p_{s+1} + \dots + p_j) \times 1$

暂定名

彼此的计算
是独立的

最优二叉查找树 —— 递推关系



■ 此时

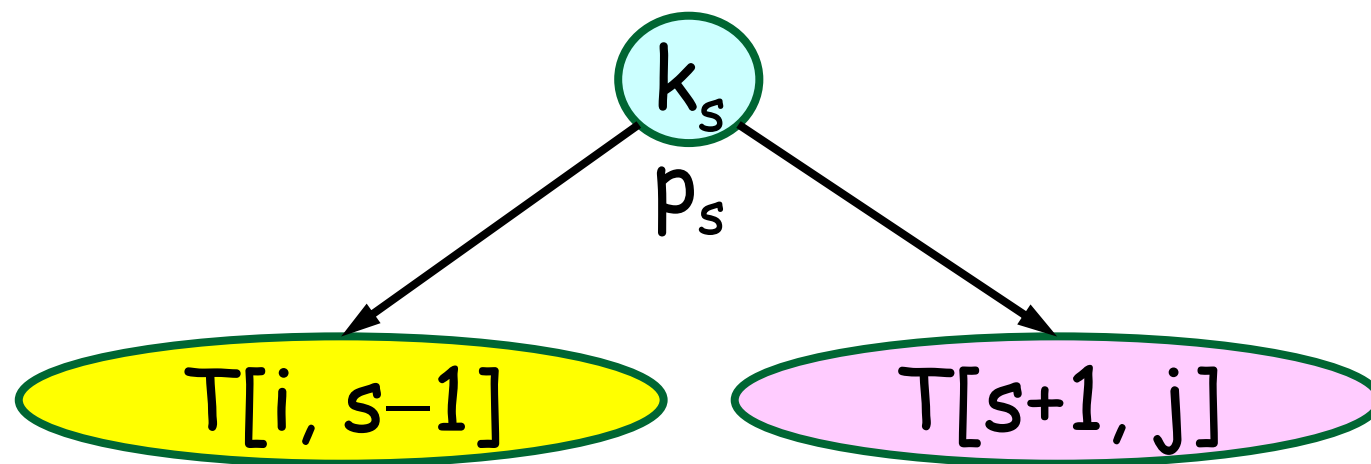
■ $C(i, j) = (p_i + \dots + p_j)$
+ $\text{cost}_{\text{left}} + \text{cost}_{\text{right}}$

彼此独立

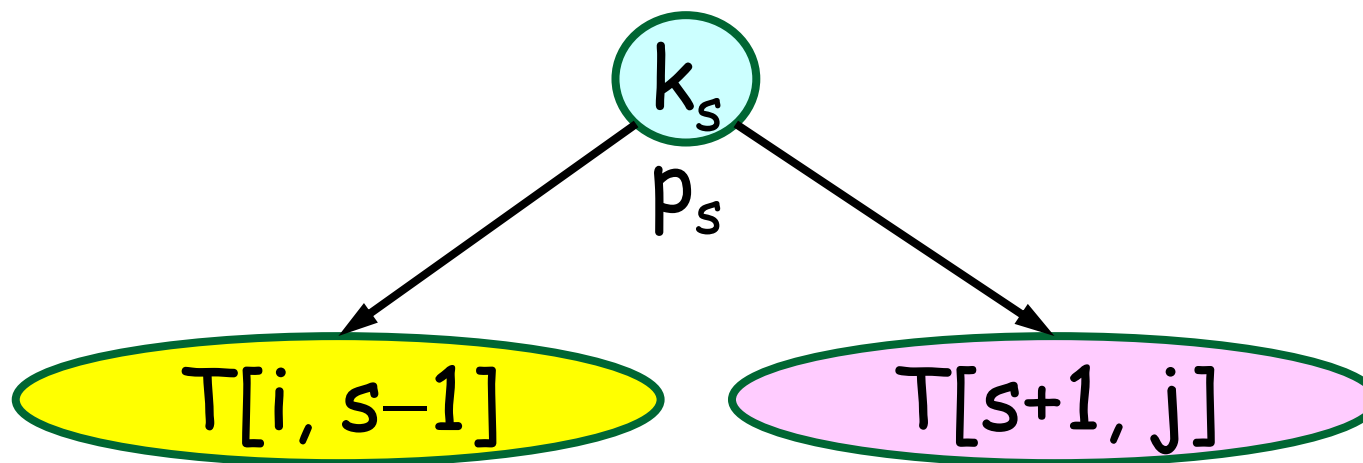
希望和达到最优（最小）值

最优二叉查找树 —— 递推关系

- 考虑 $T(i, j)$ 的根
- 如果根的键值是 k_s ，那么其左子树一定是 $T(i, s-1)$ ，右子树一定是 $T(s+1, j)$ ($i \leq s \leq j$)



最优二叉查找树 —— 递推关系



■ 此时

$$C(i, j) = C(i, s-1) + C(s+1, j) + (p_i + \dots + p_j)$$

■ 之后就是对于所有可能的 s 计算其最小值

最优二叉查找树 —— 递推关系

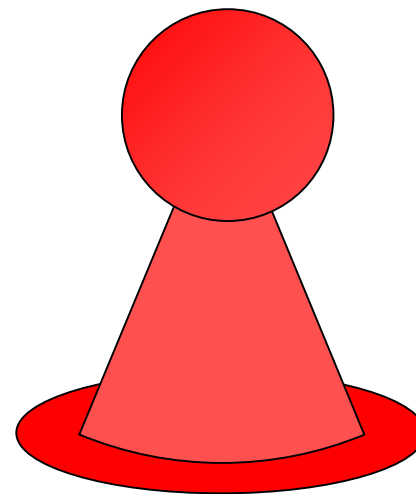
- $C(i, j) = \min\{ C(i, s-1) + C(s+1, j) + \underbrace{(p_i + \dots + p_j)} \}$
 $(i \leq s \leq j)$
- 初值 $i > j$ 时, $C(i, j) = 0$
- 令 $s(k) = p_1 + \dots + p_k, s(0) = 0$
 - 可以用 $O(n)$ 时间计算诸 $s(k)$
- 则可以使用 $s(j) - s(i-1)$ 计算 $(p_i + \dots + p_j)$

最优二叉查找树 —— 问题扩展

■ 问题描述

- 给定 n 个不同键值 k_1, k_2, \dots, k_n , 每个键值 k_i 被访问的概率为 p_i
 - 假设 $k_1 < k_2 < \dots < k_n$
- 补充键值 $x_0 = -\infty$ 和 $x_{n+1} = +\infty$, 查找概率都是0
- 待查找键值落入开区间 (k_i, k_{i+1}) 的概率为 q_i
- 于是有 $(p_1 + p_2 + \dots + p_n) + (q_0 + q_1 + \dots + q_n) = 1$
- 应如何构建二叉查找树以最小化其期望查找开销？
 - 包括查找成功和查找不成功的情况

跳棋棋盘



跳棋棋盘

- 考虑一个 $n \times n$ 的方格棋盘
- 第 i 行第 j 列的方格的开销为 $c(i, j)$
- 右图为 5×5 棋盘的示例
 $c(1, 3) = 5$

5	6	7	4	7	8
4	7	6	1	1	4
3	3	5	7	8	2
2	-	6	7	0	-
1	-	-	5*	-	-
	1	2	3	4	5

跳棋棋盘

- 假设有一个棋盘和一枚棋子
- 从最下面一行的方格 (1, 3) 开始
- 棋子每步只能沿对角线向左前方、正前方或者右前方跳一格
 - 也就是说，位于 (1, 3) 的棋子下一步只可以移动到 (2, 2)、(2, 3) 或 (2, 4)
- 希望棋子跳到最上面一行中，且途径的方格的总开销达到**最小**

5					
4					
3					
2		X	X	X	
1			O		
	1	2	3	4	5

跳棋棋盘

- 定义函数 $q(i,j)$ 为

$q(i,j)$ = 到达方格 (i,j) 的最小总开销

- 目标就是计算

$$\min_{1 \leq j \leq n} \{ q(n,j) \}$$

跳棋棋盘

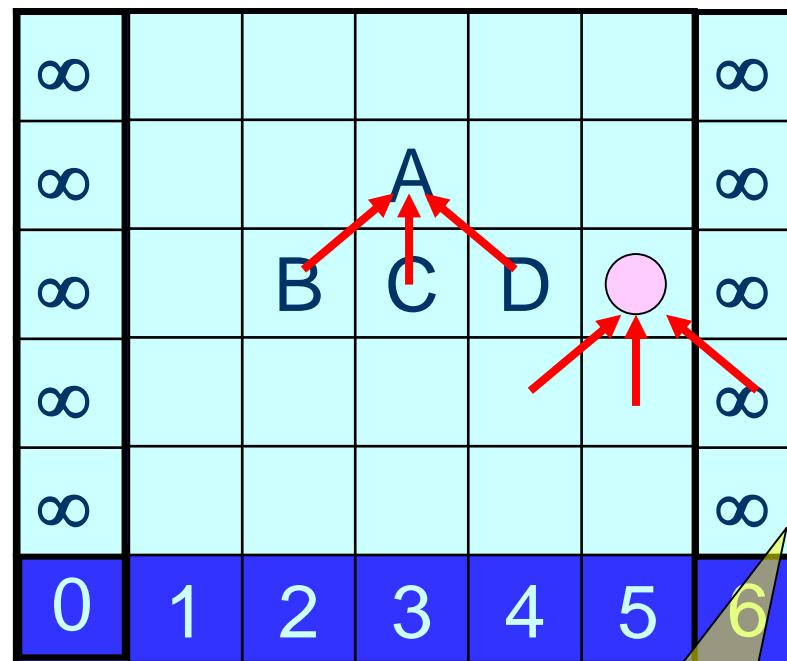
- 很容易得到:

5					
4			A		
3		B	C	D	
2					
1					
	1	2	3	4	5

$$q(A) = \min \{ q(B), q(C), q(D) \} + c(A)$$

跳棋棋盘

- 很容易得到:



$$q(A) = \min \{ q(B), q(C), q(D) \} + c(A)$$

哨兵 (sentinel) 技术

ComputeShortestPathArrays

```
1.  for  $x = 1$  to  $n$ 
2.     $q(1, x) \leftarrow \infty$ 
3.     $q(1, (n+1)/2) \leftarrow c(1, x)$ 
4.  for  $y = 1$  to  $n$ 
5.     $q(y, 0) \leftarrow \infty$ 
6.     $q(y, n + 1) \leftarrow \infty$ 
7.  for  $y = 2$  to  $n$ 
8.    for  $x = 1$  to  $n$ 
9.       $m \leftarrow \min \{ q(y - 1, x - 1), q(y - 1, x), q(y - 1, x + 1) \}$ 
10.      $q(y, x) \leftarrow m + c(y, x)$ 
11.     if  $m = q(y - 1, x - 1)$  then
12.        $p(y, x) \leftarrow -1$ 
13.     else if  $m = q(y - 1, x)$  then
14.        $p(y, x) \leftarrow 0$ 
15.     else
16.        $p(y, x) \leftarrow 1$ 
```

记录道路
Track the path

跳棋棋盘

ComputeShortestPath

1. **Call ComputeShortestPathArrays**
2. $minIndex \leftarrow 1$ // 总开销最小的列号
3. $min \leftarrow q(n, 1)$ // 总开销的最小值
4. **for** $i = 2$ **to** n
5. **if** $q(n, i) < min$ **then**
6. $minIndex \leftarrow i$
7. $min \leftarrow q(n, i)$
8. **Call PrintPath** ($n, minIndex$)

PrintPath (y, x)

1. **print** (x)
2. **print** ("<-")
3. **if** $y = 2$ **then**
4. **print** ($x + p(y, x)$)
5. **else**
6. **Call PrintPath** ($y - 1, x + p(y, x)$)

自上而下输出
各行的列号

跳棋棋盘

5	6	7	4	7	8
4	7	6	1	1	4
3	3	5	7	8	2
2	-	6	7	0	-
1	-	-	5*	-	-
	1	2	3	4	5

5	∞	24	20	12	15	16	∞
4	∞	21	18	13	8	11	∞
3	∞	14	16	12	13	7	∞
2	∞	∞	11	12	5	∞	∞
1	∞	∞	∞	5*	∞	∞	∞
	0	1	2	3	4	5	6

Min

跳棋棋盘

■ 注 1

□ “∞”

5	∞	6	7	4	7	8	∞
4	∞	7	6	1	1	4	∞
3	∞	3	5	7	8	2	∞
2	∞	∞	6	7	0	∞	∞
1	∞	∞	∞	5*	∞	1	∞
	0	1	2	3	4	5	6

错误的结果

5	∞	20	16	8	11	12	∞
4	∞	21	14	9	4	7	∞
3	∞	14	16	8	9	3	∞
2	∞	∞	11	12	1	∞	∞
1	∞	∞	∞	5*	∞	1	∞
	0	1	2	3	4	5	6

跳棋棋盘

■ 注 2

□ 也可以由上至下 (TOP DOWN)

5	∞	6	7	4	7	8	∞
4	∞	7	6	1	1	4	∞
3	∞	3	5	7	8	2	∞
2	∞		6	7	0		∞
1	∞			5*			∞
	0	1	2	3	4	5	6

5	∞	6	7	4	7	8	∞
4	∞	13	10	5	5	11	∞
3	∞	13	10	12	13	7	∞
2	∞	X	16	17	7	X	∞
1	∞	X	X	12	X	X	∞
	0	1	2	3	4	5	6

无用值

目标值

动态规划

- ① 刻画最优解的结构特性
 - $P(X)$
 - 例如 $P(n), P(n, w)$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

■ 通常而言， ϕ 是 $\max\{\}$ 或者 $\min\{\}$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

■ 示例

- 斐波那契数: $F(n) = F(n-1) + F(n-2)$
- 找零问题: $M(n) = \min \{ 1 + M(n-d_i) \}$
- 最长单调子序列: $L(i) = \max \{ L[j] + 1 \mid (1 \leq j < i, S[j] < S[i]) \}$
- 最大子段和: $C[j] = a_j \text{ or } a_j + C[j-1]$
- 背包问题: $K(n, w) = \max \{ K(n-1, W-w_n) + v_n, K(n-1, W) \}$
- 投资问题: $F(k, x) = \max \{ f(k, x_k) + F(k-1, x-x_k) \}$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

■ 示例

□ LCS

$$\blacksquare \text{Len}[X_{i-1}, Y_{j-1}] = \text{len}[X_{i-1}, Y_{j-1}] + 1 \text{ or } \max\{\text{len}[X_{i-1}, Y_j], \text{len}[X_i, Y_{j-1}]\}$$

□ SCS

$$\blacksquare \text{Len}[X_{i-1}, Y_{j-1}] = \text{len}[X_{i-1}, Y_{j-1}] + 1 \text{ or } \min\{\text{len}[X_{i-1}, Y_j] + 1, \text{len}[X_i, Y_{j-1}] + 1\}$$

□ 编辑距离

$$\blacksquare M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1])$$

动态规划

■ ② 将问题划分为子问题

$$P(\mathbf{X}) = \phi \left(f \left(P(\mathbf{X} - \mathbf{A}_1), \dots, P(\mathbf{X} - \mathbf{A}_d) \right) \right)$$

■ 示例

□ 矩阵链乘积

$$\blacksquare C[i, j] = \min_{i \leq k < j} \{ C[i, k] + C[k+1, j] + m_{i-1}m_km_j \}$$

□ 最优二叉查找树

$$\blacksquare C(i, j) = \min_{i \leq s \leq j} \{ C(i, s-1) + C(s+1, j) + (p_i + \dots + p_j) \}$$

□ 跳棋棋盘

$$\blacksquare q(A) = \min(q(B), q(C), q(D)) + c(A)$$

动态规划

■ ③ 自底而上计算



■ ④ 注意初值

动态规划的基本要素

- 一个最优化多步决策问题适合用动态规划法求解有两个要素：**最优子结构特性**和**重叠子问题**
- 最优子结构
 - 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的决策序列
 - 一个问题的最优解总是包含所有子问题的最优解
 - 但**不是**说：如果你有所有子问题的最优解，然后你可以**随便**把它们组合起来得到一个最优的解决方案

最优子结构

■ 例如 1、5 找零问题

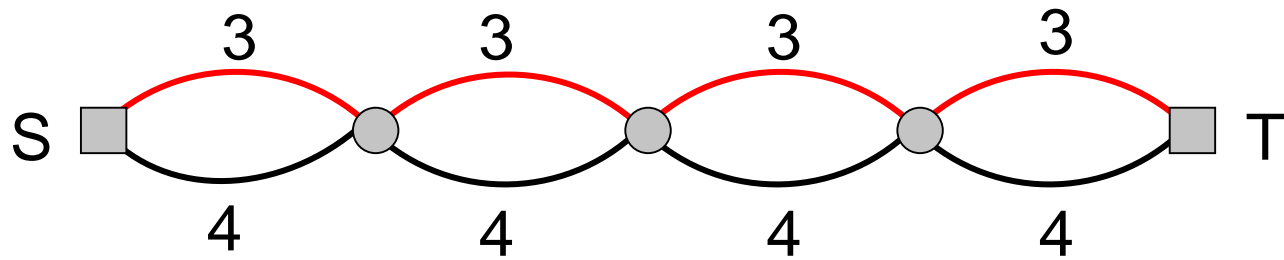
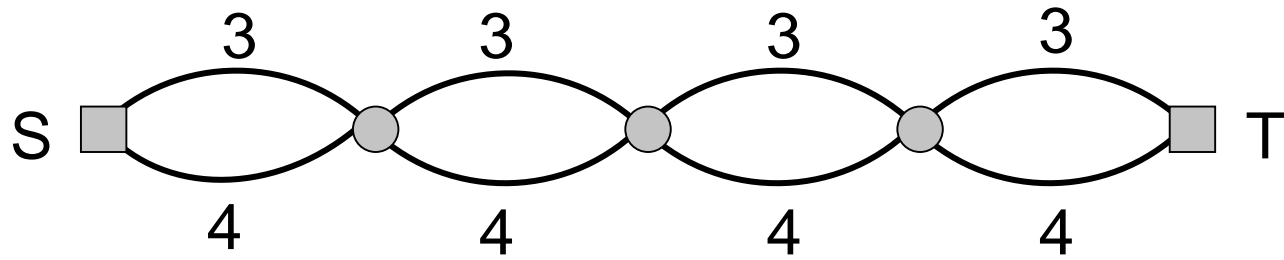
- 凑成8元的最优解是 $5 + 1 + 1 + 1$
- 凑成9元的最优解是 $5 + 1 + 1 + 1 + 1$
- 但是凑成17元的最优解并不是

$$5 + 1 + 1 + 1 + 5 + 1 + 1 + 1 + 1$$

- 然而，的确有一种方法可以把凑成17元的问题的最优解分解为子问题的最优解的组合（例如，凑成15元 + 凑成2元）
- 所以，找零问题满足最优子结构

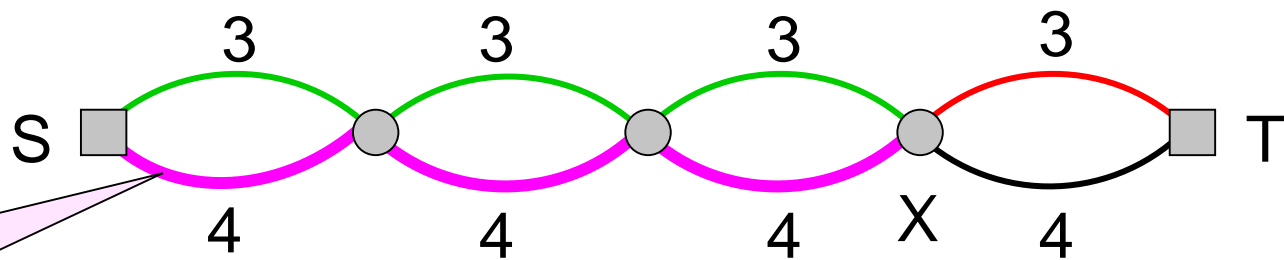
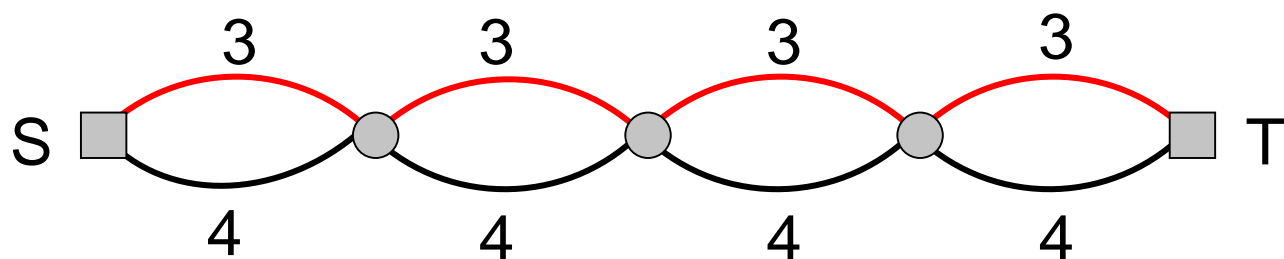
最优子结构

- 但**不是**所有问题都满足最优子结构
- 【例】 求总长模10的最短道路



最优子结构

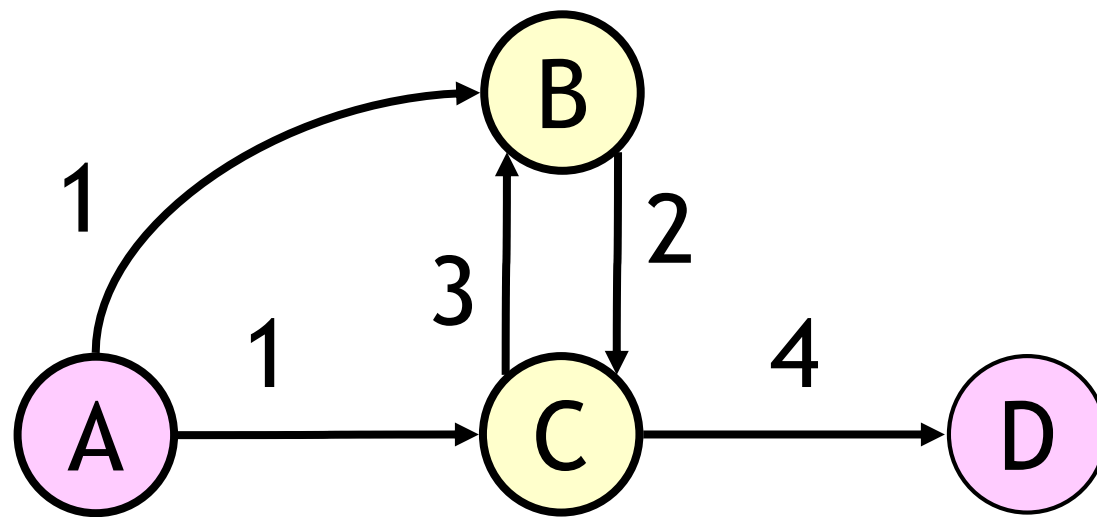
- 但**不是**所有问题都满足最优子结构
- **【例】** 求总长模10的最短道路



已然比绿色道路短
(虽然还不是最短)

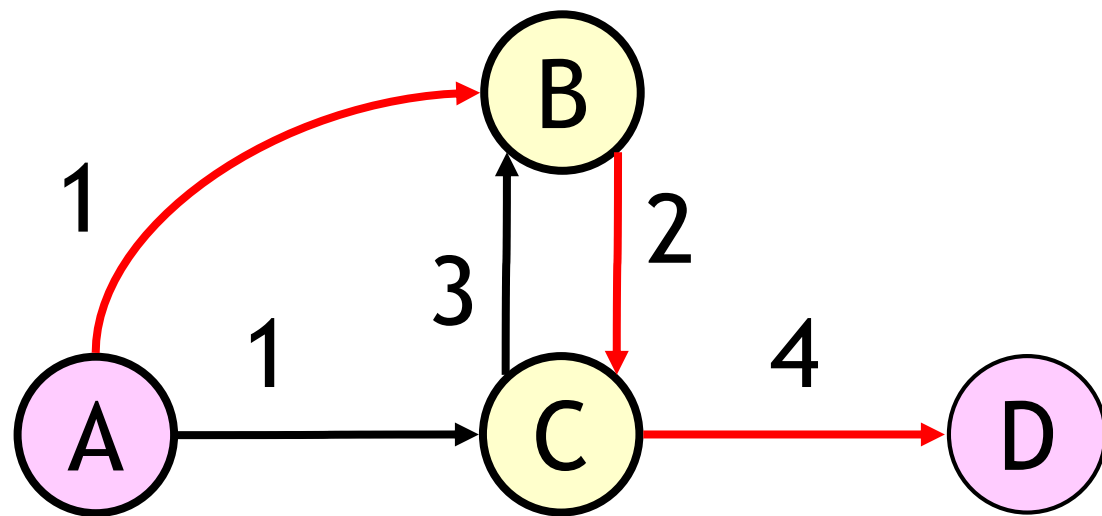
最优子结构

- 再如：最长简单道路问题
(出租车敲竹杠问题)
- 右图中，从 A 到 D 的最长简单道路是 A B C D



最优子结构

- 再如：最长简单道路问题
(出租车敲竹杠问题)
- 右图中，从A到D的最长道路是A B C D
- 但是，子道路A B不是从A到B的最长简单道路 (A C B更长)
- 这个问题不满足最优性原则
- 因此，最长简单道路问题不能用动态规划方法解决



End