Dynamic Programming

Dynamic programming: a tabular (programming) method applied to **optimization problems**.

> Divide a problem into several subproblems that are not independent (sharing subproblems). Avoid recomputing the same subproblem by solving every subproblem just once and saving the answer in a table.

Step 1. Characterize the structure of an optimal solution. (optimal substructure?)

Step 2. Recursively define the value of an optimal solution. (recurrence)

Step 3. Compute the value of an optimal solution in a ***bottom-up*** fashion. 決定填表順序

Step 4. Construct an optimal solution from the computed information. (sometimes omitted) (backtracking)

15.1 The rod-cutting problem (1-d DP)

Input:

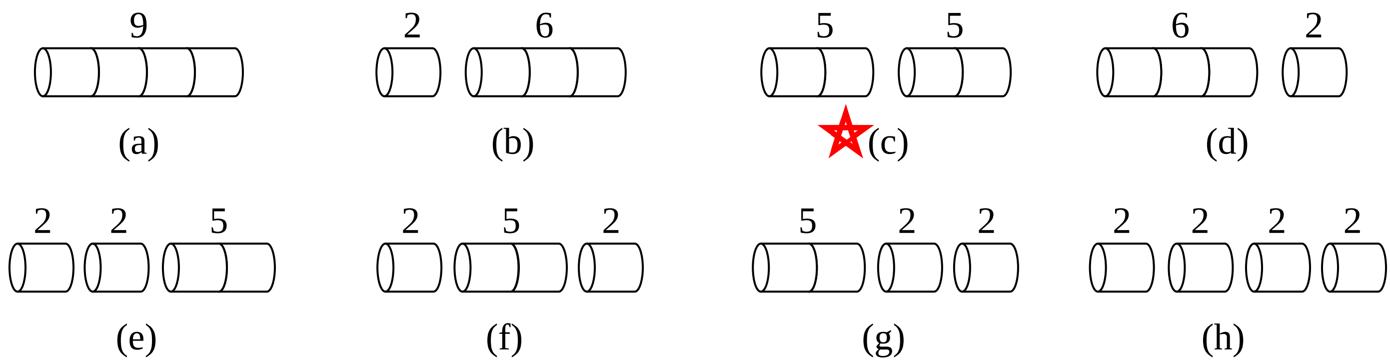
n, the length of a (steel) rod

$p[i]$, the price of a rod of length i

Output: the maximum revenue r^*

length i	1	2	3	4	5	6	7	8	9	10	11
price $p[i]$	2	5	6	9	11	16	17	20	22	24	25

A price table

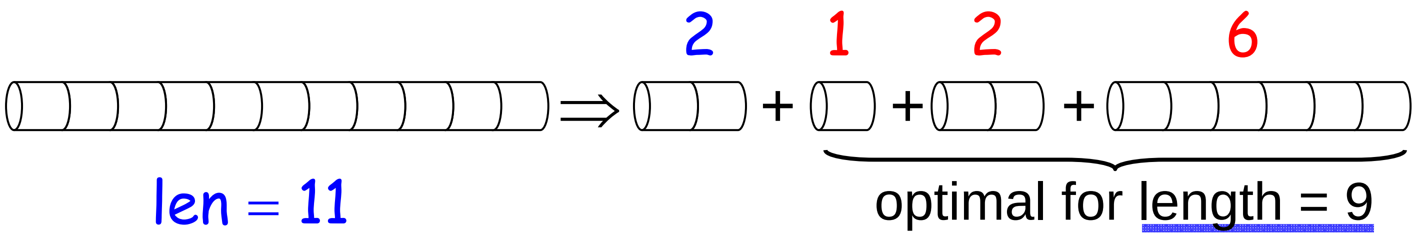


The 8 possible ways for selling a rod of length 4
((c) is optimal, where $r^* = 10$)

Step 1. An optimal solution to an instance contains optimal solutions to sub-instances.

Example:

If (2, 1, 2, 6) is optimal for length = 11,
then (1, 2, 6) is optimal for length = 9



Step 2.

Let $r[j]$ be the maximum revenue for length = j. Then

15-3a

$$r[j] = \begin{cases} 0 & \text{if } j = 0 \\ \max_{1 \le i \le j} \{ p[i] + r[j-i] \} & \text{if } j > 0\,. \end{cases}$$

The maximum revenue r^* is $r[n]$.

Step 3. Compute r and s (for **Step 4**)

15-3b

BOTTOM-UP-CUT-ROD(*p*, *n*) (bottom-up)

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n do      // compute r[j]
    r[j] ← –∞
    for i ← 1 to j do
        if r[j] < p[i] + r[j – i] then
            r[j] ← p[i] + r[j – i]
            s[j] ← i
return r and s
```

find best i
(1st cut)

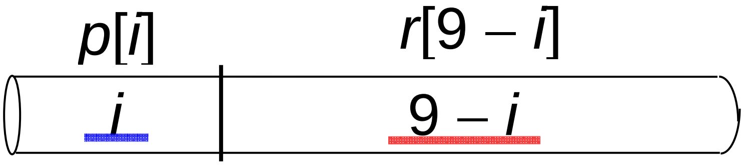
for backtracking

- $T(n)=O(n^2)$

Example: (n = 11, j = 9)

length i	1	2	3	4	5	6	7	8	9	10	11
price $p[i]$	2	5	6	9	11	16	17	20	22	24	25

i	0	1	2	3	4	5	6	7	8	9	10	11
$R[i]$	0	2	5	7	10	12	16	18	21	23	26	28
$s[i]$	0	1	2	1	2	1	6	1	2	1	2	1



first cut at i ($1 \le i \le 9$)

$$r[9] = \max\left\{\begin{array}{lll} p[1]+r[8], & p[2]+r[7], & p[3]+r[6] \\ p[4]+r[5], & p[5]+r[4], & p[6]+r[3] \\ p[7]+r[2], & p[8]+r[1], & p[9]+r[0] \end{array}\right\}$$

$$= \max\left\{\begin{array}{lll} 2+21, & 5+18, & 6+16 \\ 9+12, & 11+10, & 16+7 \\ 17+5, & 20+2, & 22+0 \end{array}\right\}$$

$= 23$ (the first cut $s[9] = 1$, 2, or 6)

* exercise: try to write a memoized version

Step 4. Using table s, by backtracking we obtain an optimal cutting in $O(n)$ time.

Example: (1, 2, 2, 6) is optimal for $n = 11$, since $s[11] = 1$, $s[10]= 2$, $s[8] = 2$, and $s[6] = 6$.

16.2 Matrix-chain multiplication (2d DP)

15-5x
15-5y

Input: $(p_0, p_1, ..., p_n)$, the dimensions of n matrices $A_1A_2 ...A_n$. (A_i is of size $p_{i-1}\times p_i$)

Output: parenthesize $A_1A_2 ...A_n$ to minimize the number of scalar multiplications.

Example: $(\overset{A_1}{p_0}, \overset{A_2}{p_1}, \overset{A_3}{p_2}, p_3) = (10, 100, 5, 50)$

$((A_1A_2)A_3) \Rightarrow 10\times100\times5 + 10\times5\times50 = 7500$ (√)

$(A_1(A_2A_3)) \Rightarrow 100\times5\times50 + 10\times100\times50 = 75000$ (×)

Step 1. An optimal solution to an instance contains optimal solutions to sub-instances.

last *

Example: if $((A_1(A_2A_3))((A_4(A_5A_6))A_7))$ is an optimal solution to $A_1A_2...A_7$, then

$(A_1(A_2A_3))$ is optimal to $A_1A_2A_3$, and
$((A_4(A_5A_6))A_7)$ is optimal to $A_4A_5A_6A_7$.

Step 2.

Let $m[i, j]$ be the minimum number of scalar multiplications for computing $A_i \ldots A_j$. We have

15-6a

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

* the best k is s[i, j]

Step 3. $m[1..n, 1..n]$ $s[1..n, 1..n]$ (for **Step 4**)

15-6b

Matrix-Chain-Order(p)

for $i \leftarrow 1$ **to** n **do** $m[i, i] = 0$ $\Rightarrow l = 1$

l matrices $\overbrace{A_i A_{i+1} \ldots A_j}$

for $l \leftarrow 2$ **to** n **do**

for $i \leftarrow 1$ **to** $n - l + 1$ **do**

$j \leftarrow i + l - 1$ find besk k (last *)

$m[i, j] = \infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_k p_j$

if $q < m[i, j]$ **then** $m[i, j] \leftarrow q$

$s[i, j] \leftarrow k$

return m and s

平 手 時 任 取

(or 記 住 全 部)

15-7x

15-6y

- $T(n)=O(n^3)$

Example: $(p_0, p_1, \ldots, p_6)=(30,35,15,5,10,20,25)$ 15-7x

k = 2 $m[2,2]+m[3,5]+p_1p_2p_5$ = $0+2500+35\times15\times20$ = 13000

k = 3 $m[2,3]+m[4,5]+p_1p_3p_5$ = $2625+1000+35\times5\times20$ = 7125

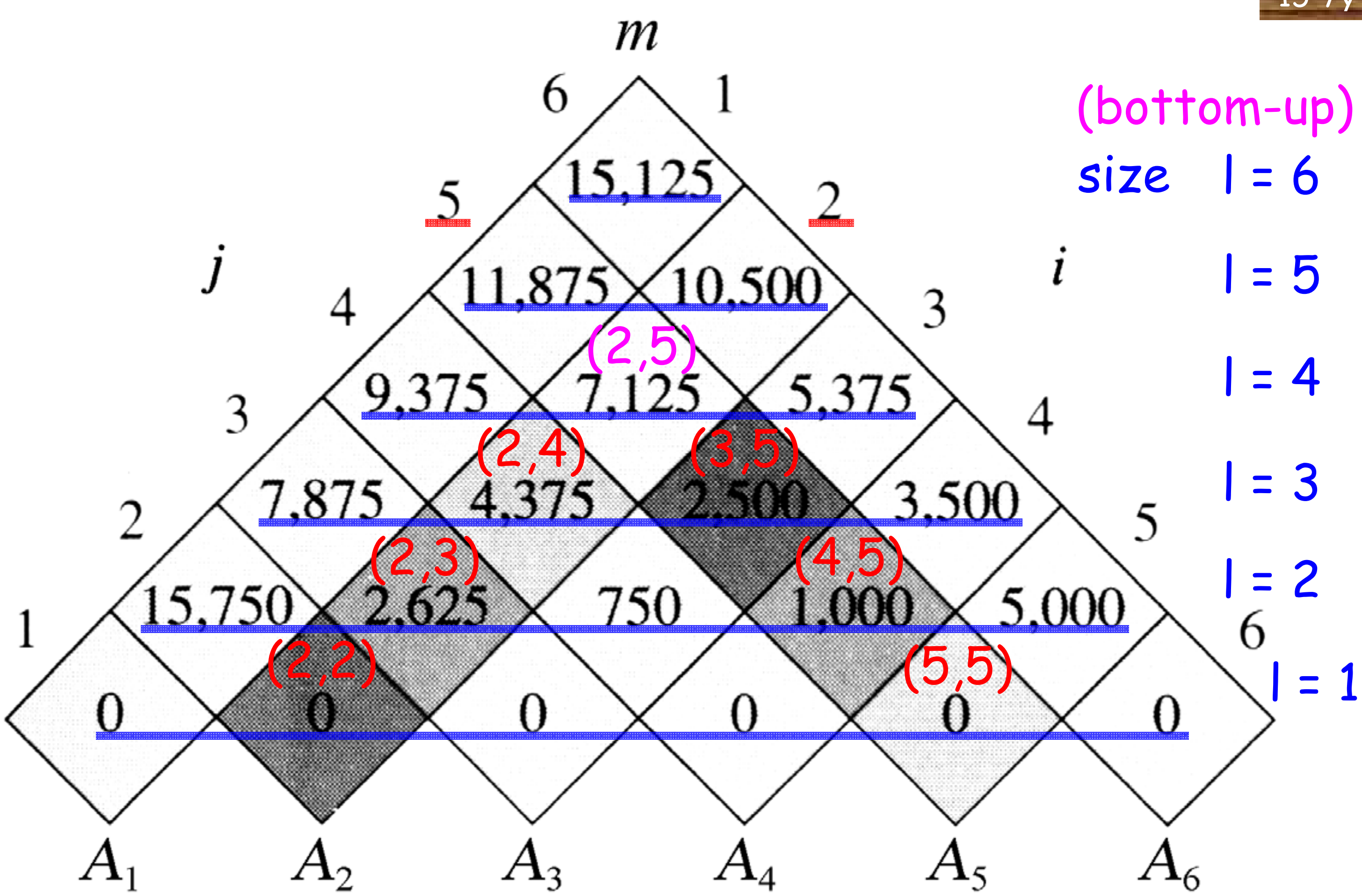
k = 4 $m[2,4]+m[5,5]+p_1p_4p_5$ = $4375+0+35\times10\times20$ = 11375

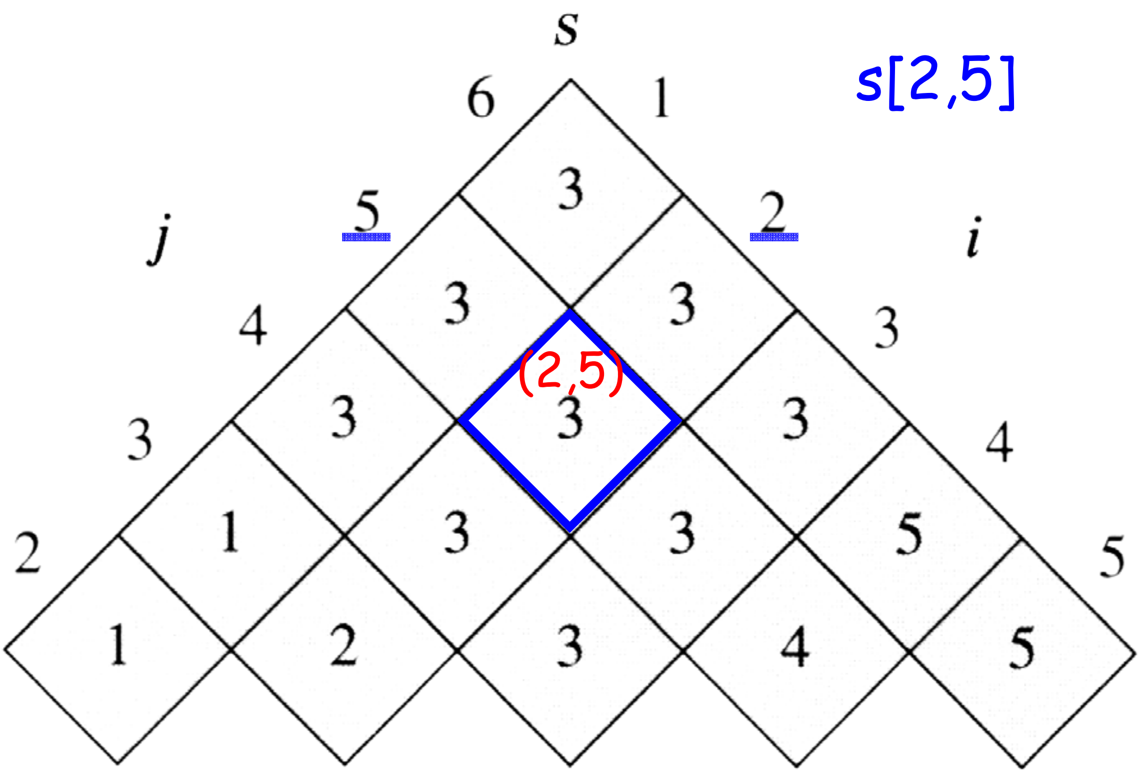
Thus, we have $m[2,5]$ = 7125 and $s[2,5]$ = 3

$A_2\ A_3\ A_4\ A_5$

k

15-7y





s[2,5]

15-8a

Step 4. Using table *s*, by backtracking we obtain $((A_1(A_2A_3))((A_4A_5)A_6))$ in $O(n)$ time.

s[1, 6] = 3

(a recursive procedure)

15.3 Elements of dynamic programming

Optimal substructure: an optimal solution to the problem contains optimal solutions to subproblems.

not independent

Overlapping subproblems: a recursive algorithm revisits the same subproblem over and over again.

top-down

Recursive-Matrix-Chain(*p*, *i*, *j*)

if $i = j$ **then return** 0

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

for $k \leftarrow i$ **to** $j - 1$ do

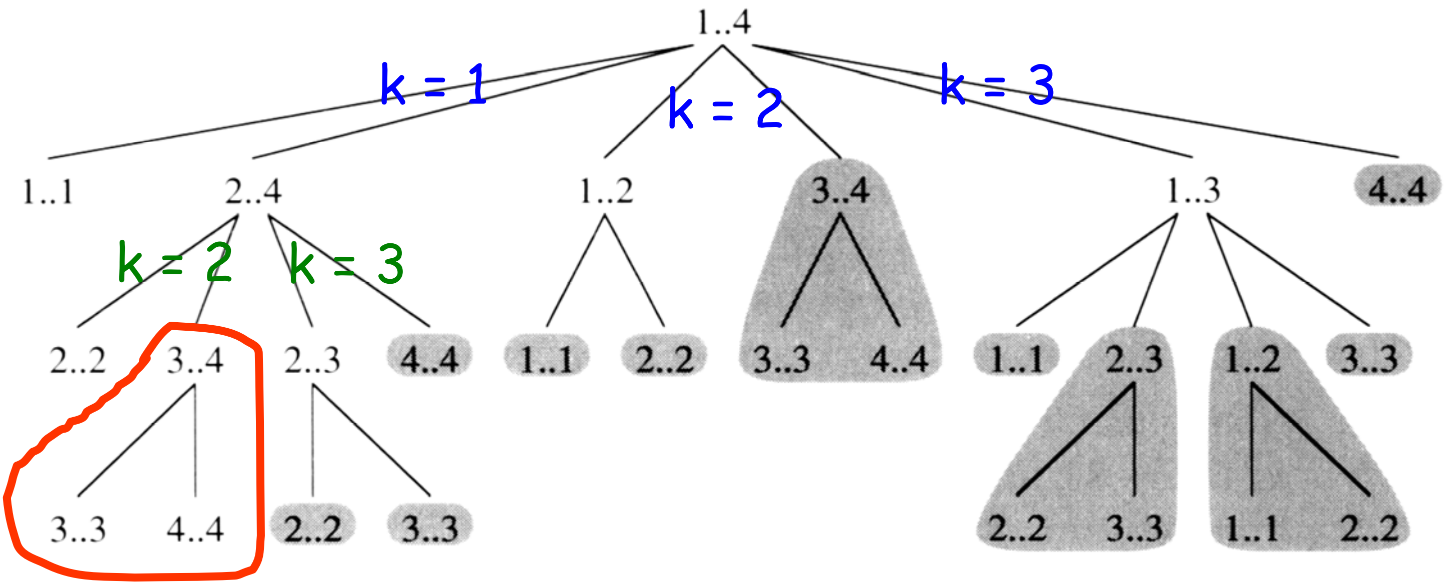
$q \leftarrow$ Recursive-Matrix-Chain(p, i, k)

+ Recursive-Matrix-Chain(p, k+1, j)

+ $p_{i-1}p_kp_j$

if $q < m[i, j]$ **then** $m[i, j] \leftarrow q$

return $m[i, j]$



- $T(n) \geq \sum_{1 \leq k \leq n-1}(T(k) + T(n-k) + 1)$

$\geq 2\sum_{1 \leq i \leq n-1} T(i) + n$

$= \Omega(2^n)$ (by substitution method)

(or Knuth's approach)

for cases when it is hard to "bottom-up" (eg. 3-D, 4-D)

Memoization:

a variation of dynamic programming (top-down)

Memoized-Matrix-Chain(*p*)

    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow i$ **to** n **do** $m[i, j] = \infty$ un-computed
    return Lookup-Chain(m, p, 1, n)

Lookup-Chain(*m*, *p*, *i*, *j*)

    if $m[i, j] < \infty$ **then return** $m[i, j]$ avoid recomputing
    if $i = j$ **then** $m[i, j] \leftarrow 0$ save the answer
        else
            for $k \leftarrow i$ **to** $j - 1$ **do**
                $q \leftarrow$ Lookup-Chain(m, p, i, k)
                    + Lookup-Chain(m, p, k+1, j)
                    + $p_{i-1}p_kp_j$
                if $q < m[i, j]$ **then** $m[i, j] \leftarrow q$ save the answer
    return $m[i, j]$

Compute & Save

- $T(n)=O(n^3)$

* Try to write a memoized recursive algorithm for the rod cutting problem.

15.4 Longest common subsequence (LCS)

Subsequence: *Z* is a subsequence of *X* iff *Z* can be obtained from *X* by deleting some characters.

Common subsequence:

$X = x_1x_2...x_7$ = abcbdab $Y = y_1y_2...y_6$ = bdcaba

common sequences: ba, bca, bcba, bdab

Longest common subsequence: bcba, bdab

Step 1. Optimal substructure 15-11a

X[1..6]

Example: $X[1..m]$ = a b c b d a b d
$Y[1..n]$ = b d c a b a c

Y[1..4]

From (b, d, a, b) = $LCS(X, Y)$, we conclude that (b, d, a) = $LCS(X[1..m-2], Y[1..n-3])$.

Step 2. 15-11a

Let $Z[1..k] = LCS(X[1..m], Y[1..n])$.

(1) If $x_m = y_n$, then

$x_m = y_n = z_k$ and

$Z[1..k-1] = LCS(X[1..m-1], Y[1..n-1])$.

(2) If $x_m \neq y_n$, then either

$Z[1..k] = LCS(X[1..m-1], Y[1..n])$ or

$Z[1..k] = LCS(X[1..m], Y[1..n-1])$

X $\boxed{1 \sim i}$ y $\boxed{1 \sim j}$

Let $c[i, j]$ be the length of $LCS(X[1..i], Y[1..j])$

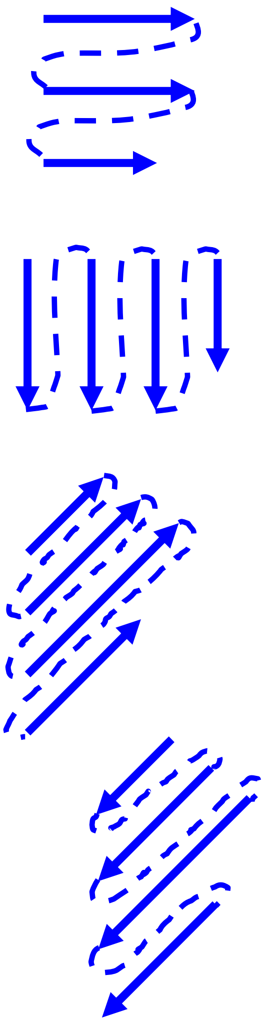
null string

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

1 2 3

Step 3. $c[0..n, 0..n]$, $b[0..n, 0..n]$ (for **Step 4**)

15-12x



i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- Time: $O(mn)$ Space: $O(mn)$
- If Step 4 is omitted, c only needs two rows. (or columns)

space: O(min{m,n})

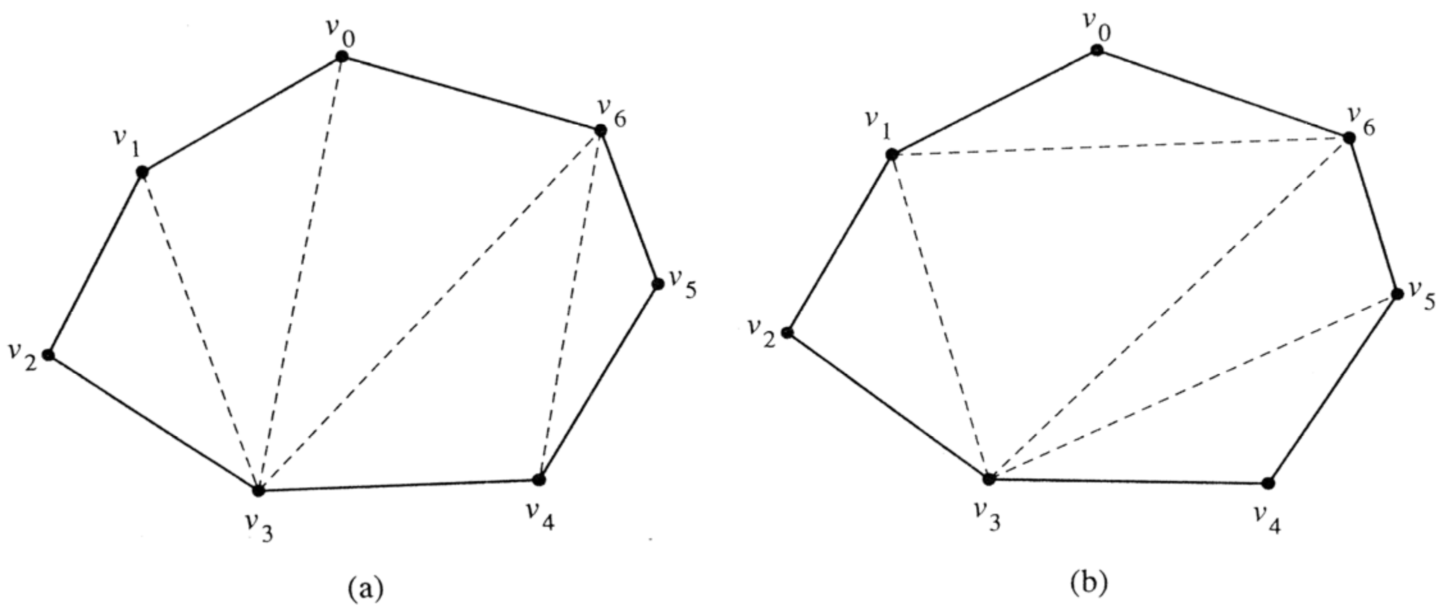
Step 4. Using table b, by backtracking we obtain $LCS(X, Y)$ = bcba in $O(m + n)$ time.

15.5 optimal binary search trees (extra class)

* Optimal polygon triangulation

Input: a convex polygon $P = (v_0, v_1, \ldots, v_{n-1})$
a cost function $w(\Delta v_i v_j v_k)$

Output: an optimal triangulation



minimize 虛線總長

- Usually, $w(\Delta v_i v_j v_k)$ is $|v_i v_j| + |v_j v_k| + |v_i v_k|$.

15-14x

Step 2. Let $t[i, j]$ be the weight of an optimal triangulation of polygon $(v_{i-1}, v_i, \ldots, v_j)$.

☆

$$t[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{t[i,k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j\} & \text{if } i < j \end{cases}$$

* t[1, n-1] is the solution !

Step 3. Similar to Step 3 of matrix chain.

15-14z

Time: $O(n^3)$ Space: $O(n^2)$

Homework: Ex. 15.2-2, 15.4-3 15.4-5, Prob. 15-3, 15-4, 15-5, 15-9

* Every problem is worth studying !

15supp-a