Ch 15. Dynamic Programming

Dynamic Programming

- 표(table)를 만들어 채워가면서 답을 구하는 방법
- Divide and Conquer 와의 차이점 : overlaps in subproblems
- meaning of “programming” here : tabular method
- used in solving optimization problem
  - find an optimal solution, as opposed to the optimal solution

Dynamic Programming

1. 최적해의 구조적 특징을 찾는다.
2. 최적해의 값을 재귀적으로 정의한다.
3. 최적해의 값을 일반적으로 상향식 방법으로 계산한다.
4. 계산된 정보들로부터 최적해를 구성한다.

examples of dynamic programming

15.1 rod cutting

15.2 matrix-chain multiplication

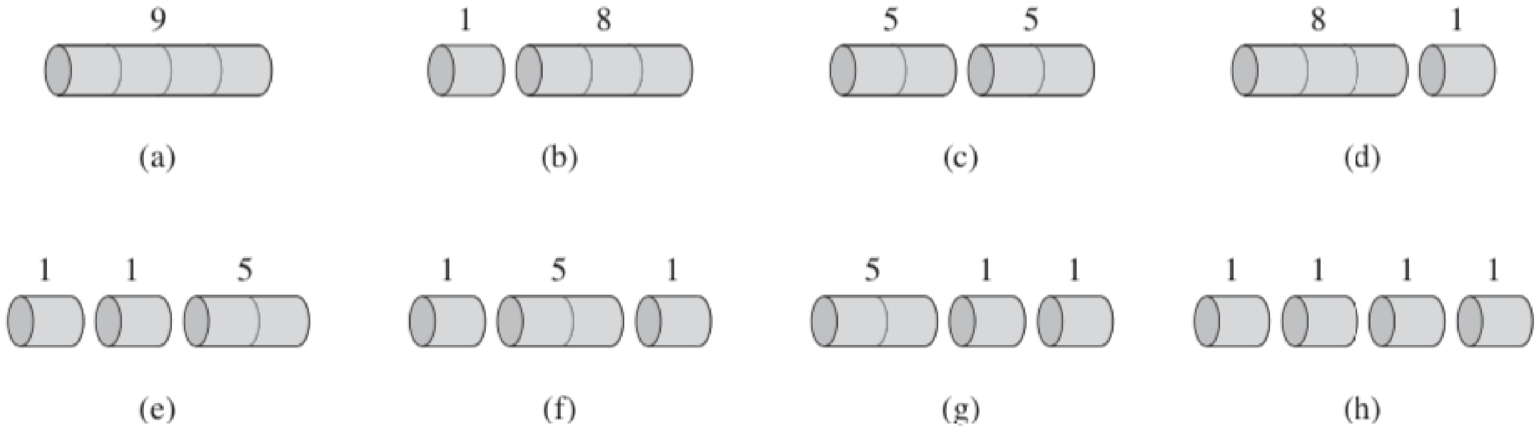
15.4 longest common subsequence

15.1 : rod cutting

- n 인치 막대를 잘라서 판매하여 얻을 수 있는 최대 수익 r_n 을 찾아라.
- 막대를 자르는 비용은 0
- sample price table

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

- 예를 들어, 4 inch rod 를 자르는 방법은 8=2^3 가지가 있다.



n-inch rod cutting

- 자르는 방법은 2^n 가지가 있다.
- 7 = 2+2+3 로 자르면 수익은 $r_7 = 5 + 5 + 8 = 18$

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

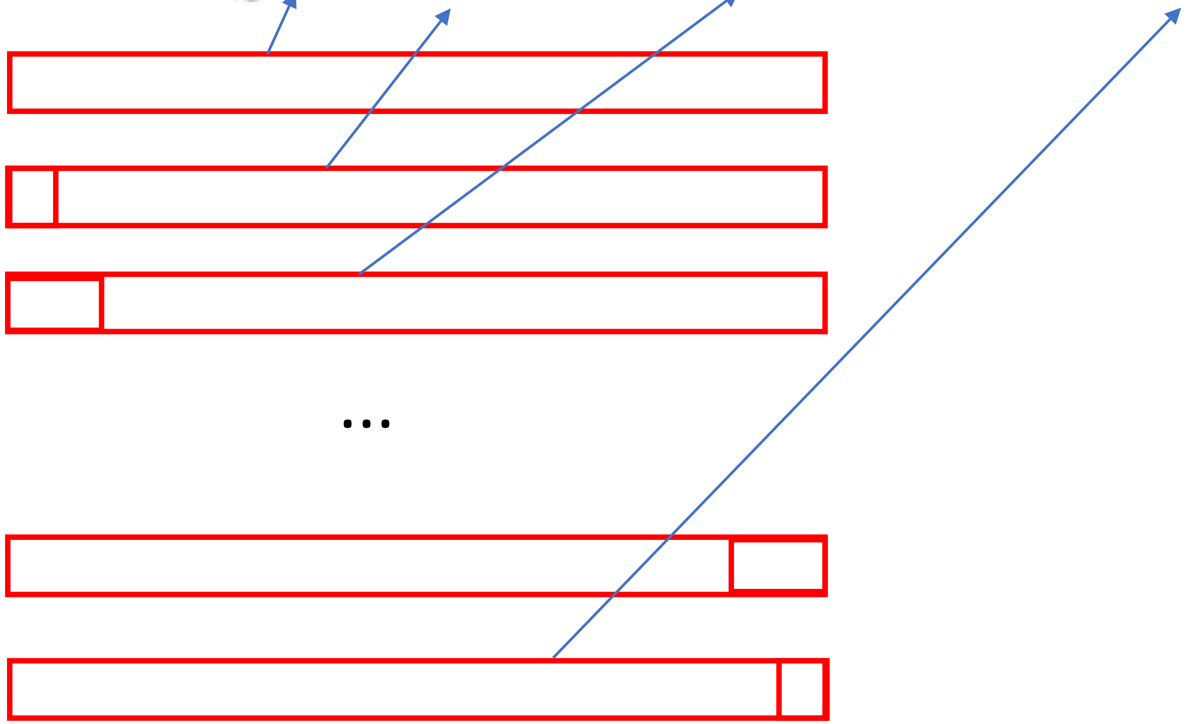
$n = i_1 + i_2 + \cdots + i_k$ 일 때 $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

p_j 는

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

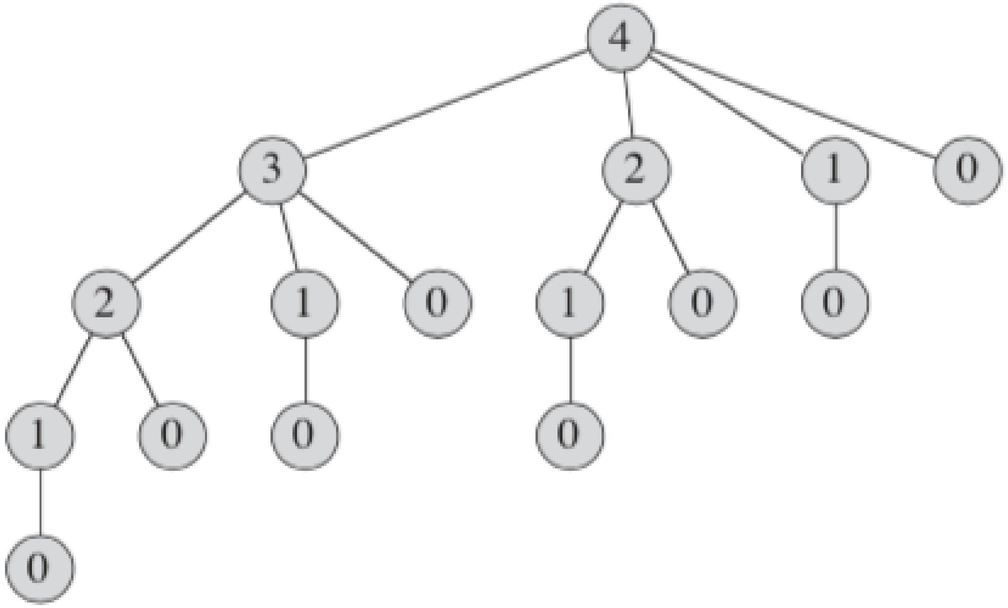
r_i for $i<n$ 으로부터 r_n 을 구할 수 있다.
→ optimal substructure 를 가졌다.

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \quad \Rightarrow \quad r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$



Recursive top-down implementation

```
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
```



$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)\,. \quad \rightarrow T(n) = 2^n$$

Dynamic Programming – top-down

```
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)
```

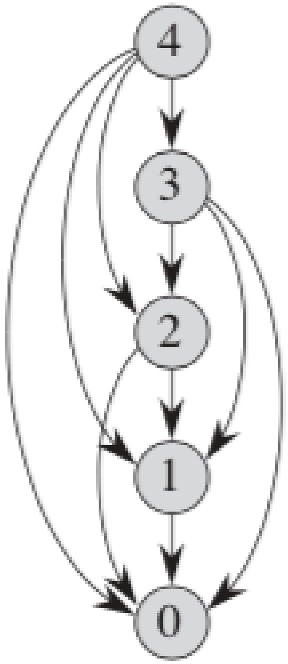
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

$\Theta(n^2)$

Dynamic Programming – bottom-up

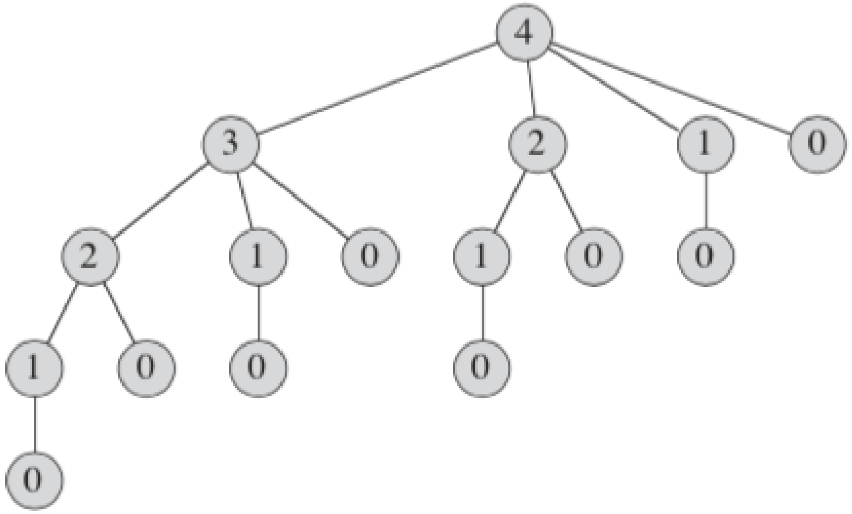
```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

$\Theta(n^2)$



subproblem graph

vs.



Reconstructing a solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

```
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]
```

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

15.2 Matrix-chain multiplication

- 여러 개의 행렬을 곱할 때 곱셈 순서에 따라 연산 갯수가 달라진다.

because

```
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
```

$A_1 : 2\times3 \quad A_2 : 3\times5 \quad A_3 : 5\times6 \rightarrow A_1A_2A_3 : 2\times6$

$(A_1A_2)A_3 : 2\times3\times5 + 2\times5\times6 = 90$

$A_1(A_2A_3) : 3\times5\times6 + 2\times3\times6 = 126$

$=\Theta(A.rows \times B.columns \times A.columns)$

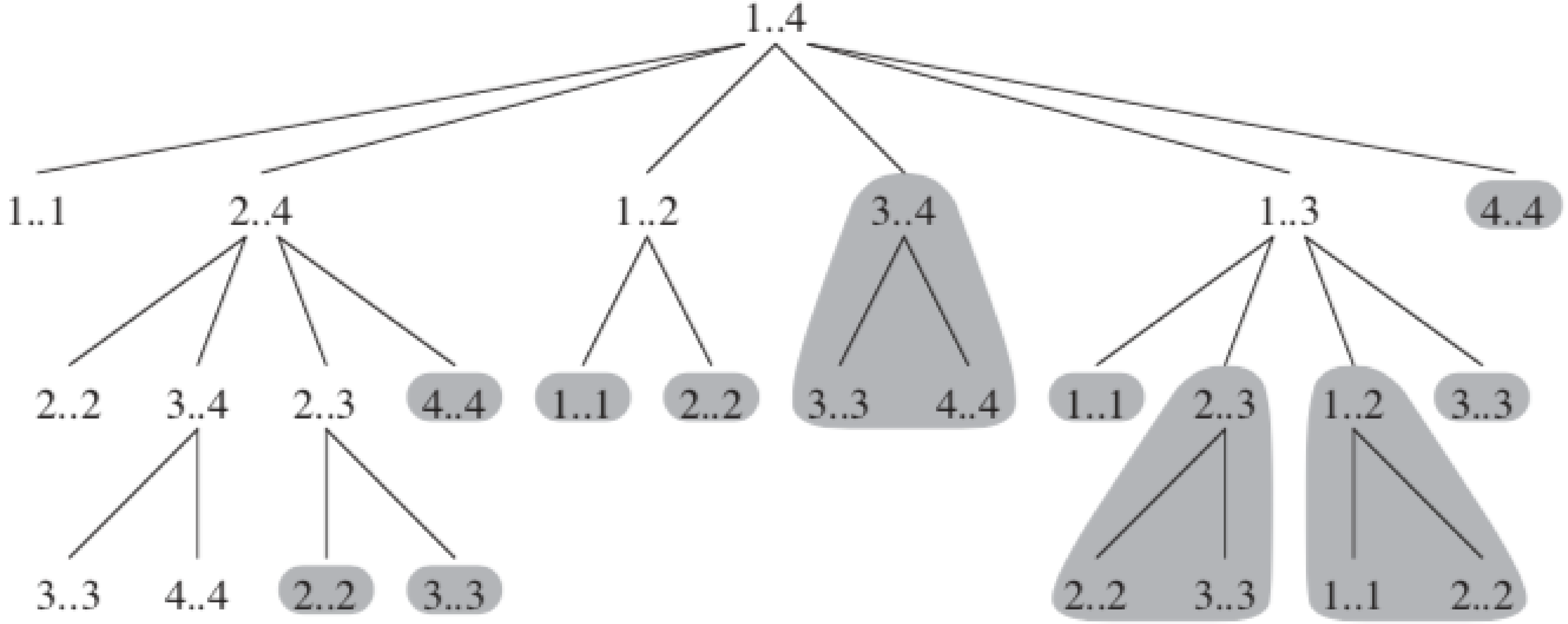
행렬 곱셈의 순서를 정하는 문제 (곱셈을 하는 게 아님)

- Exhaustive search when n = 4

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \displaystyle\sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n \geq 2\,. \end{cases}$$

$= \Omega(2^n)$

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$



Dynamic Programming

1. 최적해의 구조적 특징을 찾는다.
2. 최적해의 값을 재귀적으로 정의한다.

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

$m[i,j]$: A_i x A_j 의 곱을 optimal 순서로 곱했을 때 연산의 횟수

p_k : A_k 의 column 의 갯수 = A_{k+1} 의 row 의 갯수

Dynamic Programming

3. 최적해의 값을 일반적으로 상향식 방법으로 계산한다.

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

을 recursive call 로 구현하면 $\Omega(2^n)$

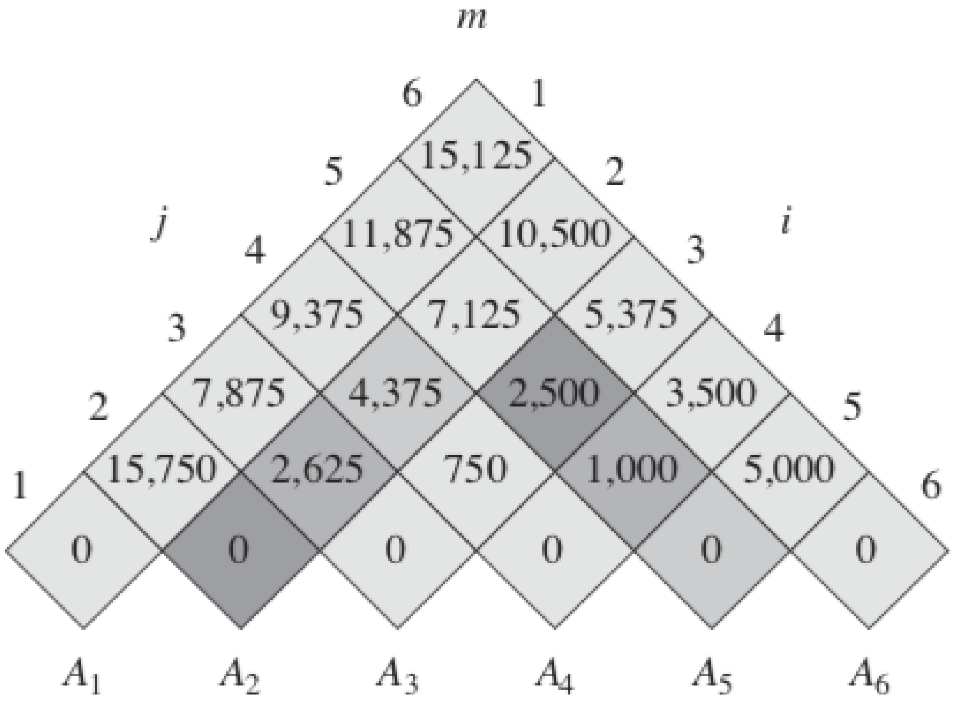
- optimal substructure 를 가지고 subproblem 들이 overlapped 되어있다.

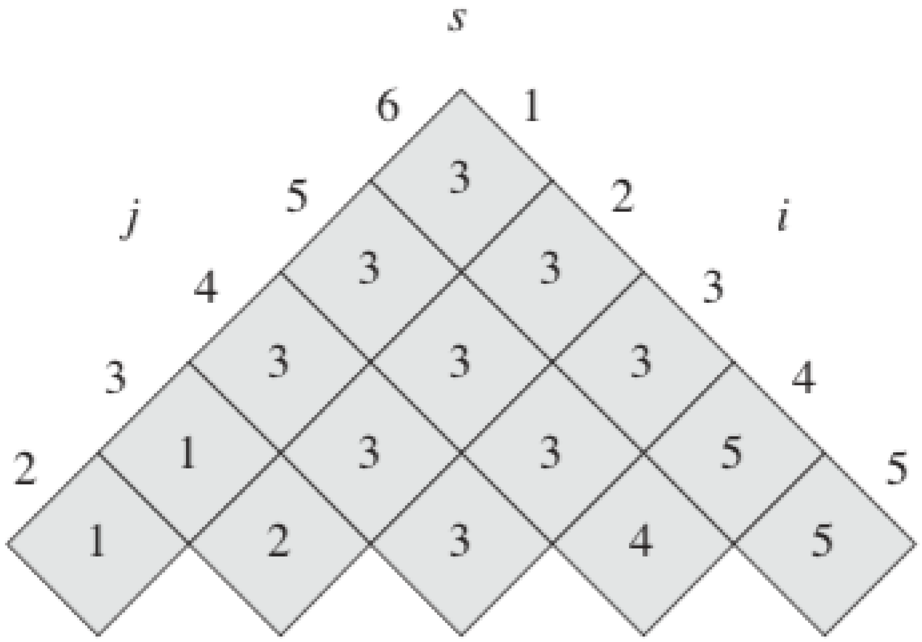
→ dynamic programming 의 조건

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            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
return m and s
```

$= O(n^3)$

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





Dynamic Programming

4. 계산된 정보들로부터 최적해를 구성한다.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
        print "A"_i
else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```

15.4 Longest common subsequence (LCS)

- subsequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ of sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$

단조 증가하는 x 의 인덱스 시퀀스 $\langle i_1, i_2, \ldots, i_k \rangle$ such that $x_{i_j} = z_j$ 가 있다.

i.e. X = <A,B,C,B,D,A,B>

의 subsequence Z = <B,C,D,B> for <2,3,5,7>

- common subsequence Z of X and Y : Z is subsequence of X, and of Y.

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

→

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

LCS-LENGTH(X, Y)

```
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
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] = "↖"
        elseif 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
```

$= \Theta(mn)$

step 4. constructing LCS

```
PRINT-LCS(b, X, i, j)
    if i == 0 or j == 0
        return
    if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
    elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
    else PRINT-LCS(b, X, i, j − 1)
```

= $O(m+n)$

	j	0	1	2	3	4	5	6
i		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

15.3 elements of dynamic programming

- maximum subarray 나 matrix multiplication 을 dynamic programming 으로 풀 수 있는가?

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one ele
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```