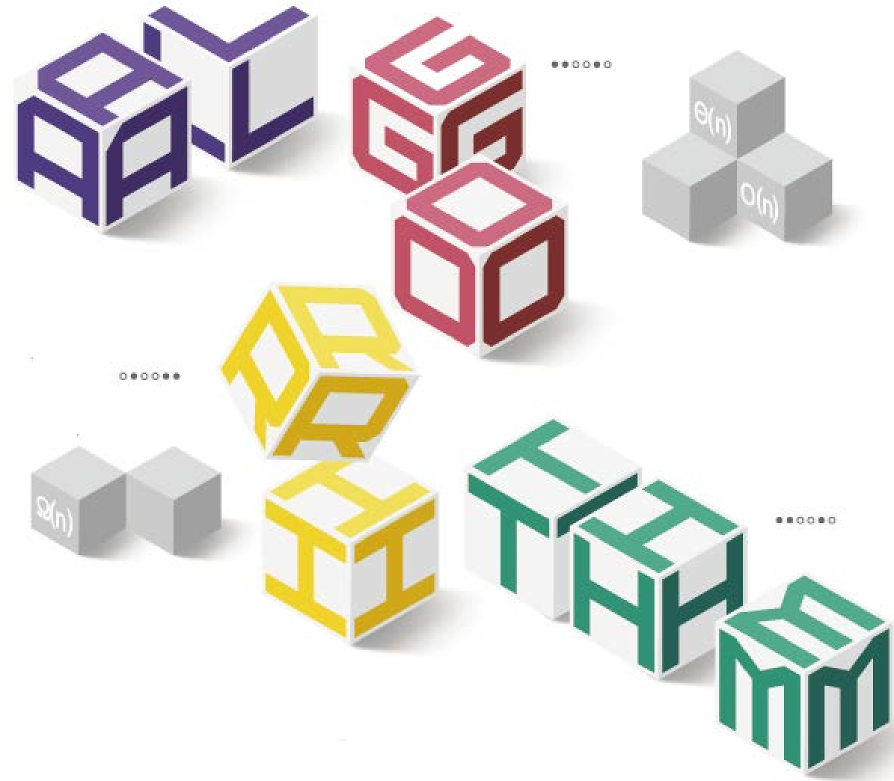


관계 중심의 사고법

쉽게 배우는 알고리즘

9장. 동적 프로그래밍 Dynamic Programming (DP)

9장. 동적 프로그래밍
Dynamic Programming (DP)

학습목표

- 동적 프로그래밍이 무엇인지 이해한다.
- 어떤 특성을 가진 문제가 동적 프로그래밍의 적용 대상인지 감지할 수 있도록 한다.
- 기본적인 몇 가지 문제를 동적 프로그래밍으로 해결할 수 있도록 한다.

배경

- 재귀적 해법
  - 큰 문제에 닮음꼴의 작은 문제가 깃든다
  - 잘쓰면 보약, 잘못쓰면 맹독
    - 관계중심으로 파악함으로써 문제를 간명하게 볼 수 있다
    - 재귀적 해법을 사용하면 심한 중복 호출이 일어나는 경우가 있다

재귀적 해법의 빛과 그림자

- 재귀적 해법이 바람직한 예
  - 퀵정렬, 병합정렬 등의 정렬 알고리즘
  - 계승(factorial) 구하기
  - 그래프의 DFS
  - …
- 재귀적 해법이 치명적인 예
  - 피보나치수 구하기
  - 행렬곱셈 최적순서 구하기
  - …

도입문제: 피보나치수 구하기

- $f(n) = f(n\text{-}1) + f(n\text{-}2)$

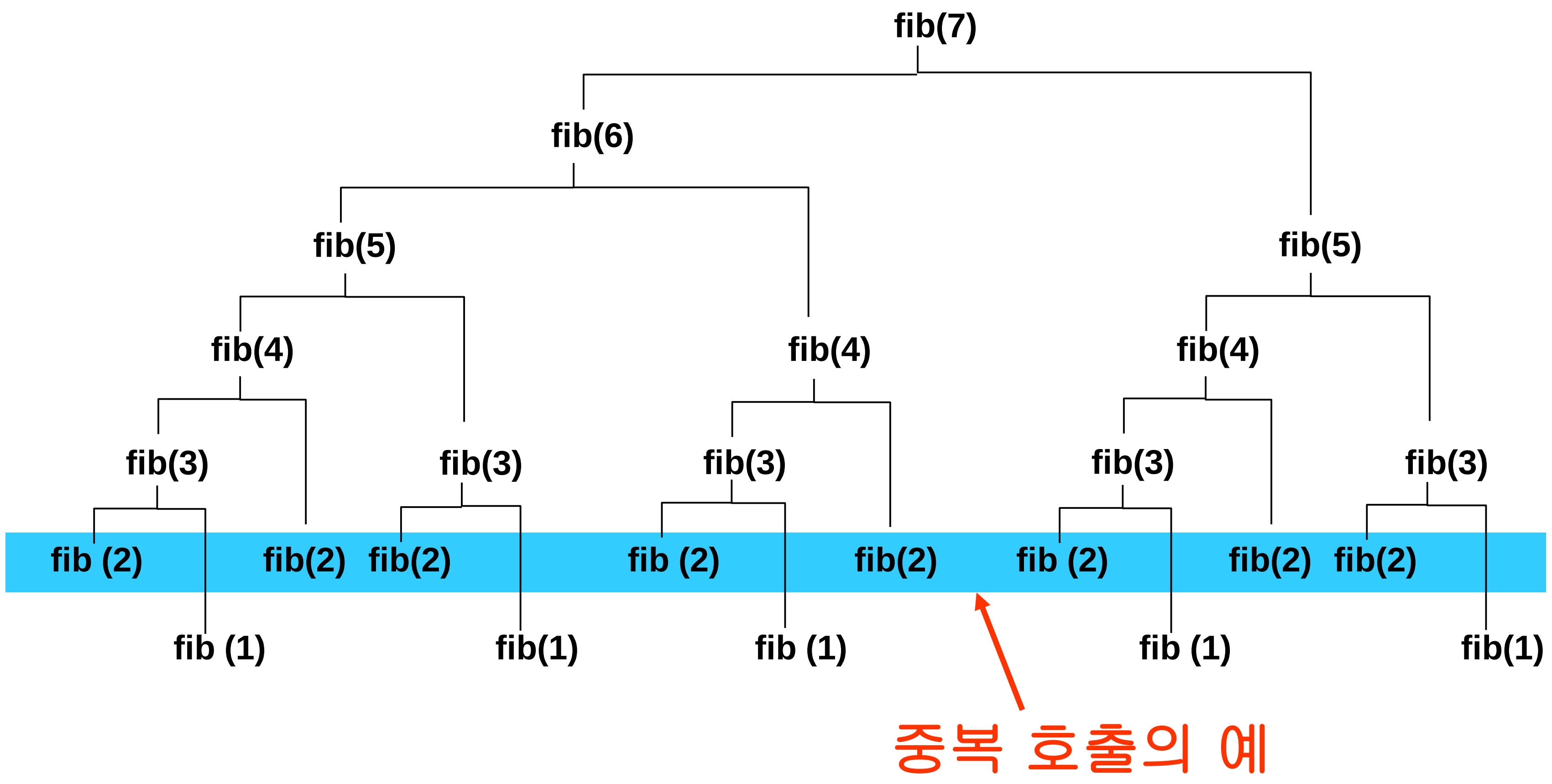
  $f(1) = f(2) = 1$
- 아주 간단한 문제지만
  - 동적 프로그래밍의 동기와 구현이 다 포함되어 있다

피보나치수를 구하는 재귀 알고리즘

```
fib(n)
{
    if (n = 1 or n = 2)
        then return 1;
        else return (fib(n-1) +fib(n-2));
}
```

✓ 엄청난 중복 호출이 존재한다

피보나치 수열의 호출 트리



피보나치수를 구하는 동적 프로그래밍 알고리즘

```
fibonacci(n)
{
      f[1] ← f[2] ← 1;
      for i ← 3 to n
            f[i] ← f[i-1] +f[i-2];
      return f[n];
}
```

✓ 선형시간에 끝난다

동적 프로그래밍의 적용 요건

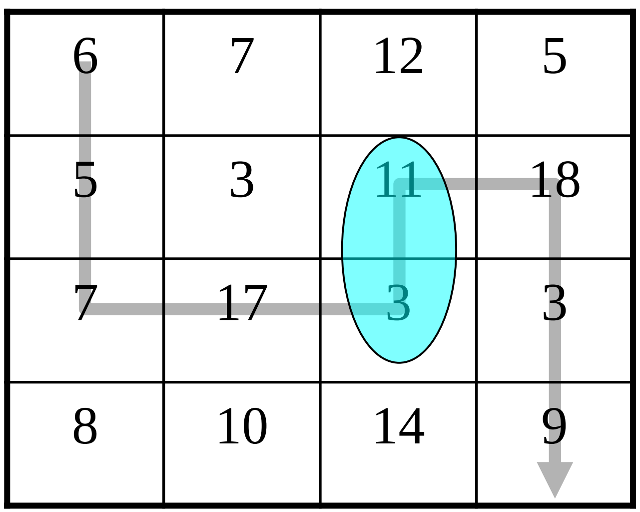
- 최적 부분구조optimal substructure
  - 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함됨
- 재귀호출시 중복overlapping recursive calls
  - 재귀적 해법으로 풀면 같은 문제에 대한 재귀호출이 심하게 중복됨

➡ 동적 프로그래밍이 그 해결책!

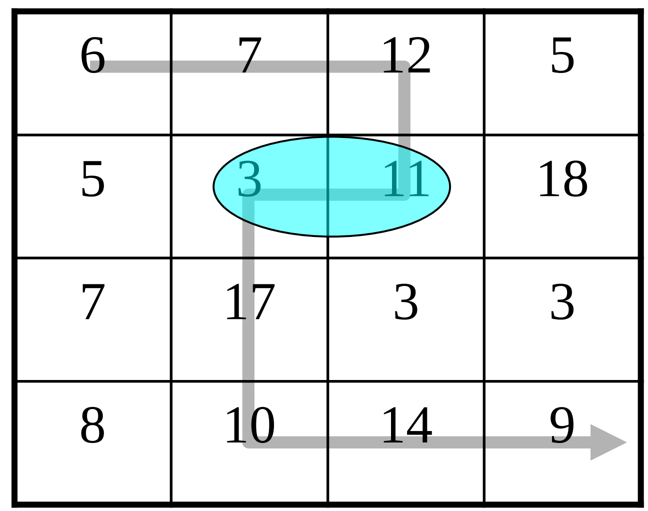
문제예 1: 행렬 경로 문제

- 양수 원소들로 구성된 $n \times n$ 행렬이 주어지고, 행렬의 좌상단에서 시작하여 우하단까지 이동한다
- 이동 방법 (제약조건)
  - 오른쪽이나 아래쪽으로만 이동할 수 있다
  - 왼쪽, 위쪽, 대각선 이동은 허용하지 않는다
- 목표: 행렬의 좌상단에서 시작하여 우하단까지 이동하되, 방문한 칸에 있는 수들을 더한 값이 최대화되도록 한다

불법 이동의 예



불법 이동 (상향)



불법 이동 (좌향)

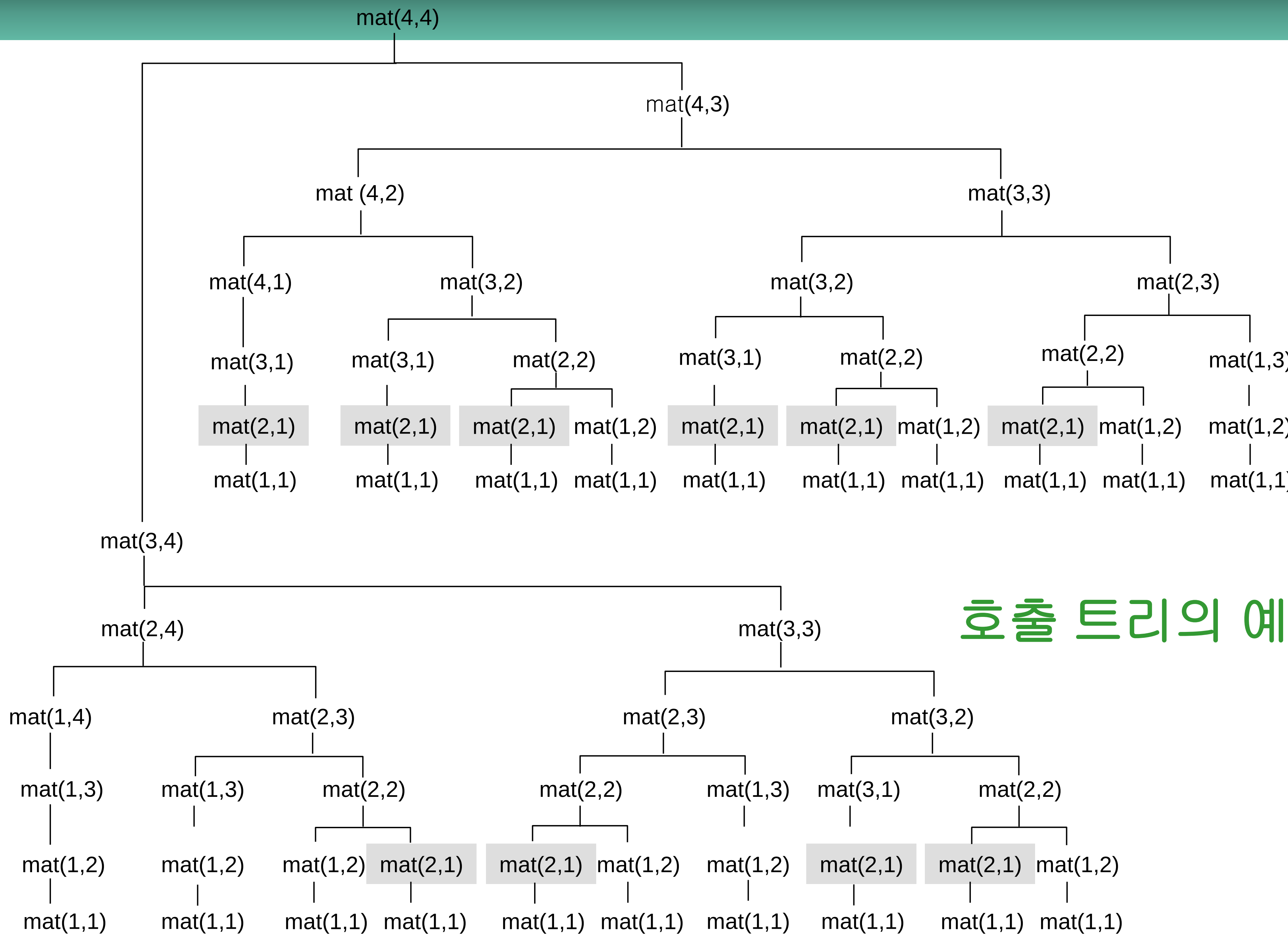
유효한 이동의 예

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

재귀 알고리즘

```
matrixPath(i, j)
▷ (i, j)에 이르는 최고점수
{
        if (i = 0 or j = 0) then return 0;
        else return (m_ij + (max(matrixPath(i-1, j), matrixPath(i, j-1))));
}
```


mat(4,4)
mat(4,3)
mat (4,2)
mat(3,3)
mat(4,1)
mat(3,2)
mat(3,2)
mat(2,3)
mat(3,1)
mat(3,1)
mat(2,2)
mat(3,1)
mat(2,2)
mat(2,2)
mat(1,3)
mat(2,1)
mat(2,1)
mat(2,1)
mat(1,2)
mat(2,1)
mat(2,1)
mat(1,2)
mat(2,1)
mat(1,2)
mat(1,2)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(3,4)
mat(2,4)
mat(3,3)
mat(1,4)
mat(2,3)
mat(2,3)
mat(3,2)
mat(1,3)
mat(1,3)
mat(2,2)
mat(2,2)
mat(1,3)
mat(3,1)
mat(2,2)
mat(1,2)
mat(1,2)
mat(1,2)
mat(2,1)
mat(2,1)
mat(1,2)
mat(1,2)
mat(2,1)
mat(2,1)
mat(1,2)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)
mat(1,1)

호출 트리의 예

DP 점화식

$c[i, j]$: (1,1)에서 (i, j)에 이르는 경로의 최대값

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ m_{ij} + \max(c[i-1, j], c[i, j-1]) & \text{otherwise.} \end{cases}$$

DP 알고리즘

```
matrixPath(n)
▷ (n, n)에 이르는 최고점수
{
    for i ← 0 to n
        c[i, 0] ← 0;
    for j ← 1 to n
        c[0, j] ← 0;
    for i ← 1 to n
        for j ← 1 to n
            c[i, j] ← m_ij + max(c[i-1, j], c[i, j-1]);
    return c[n, n];
}
```

문제예 2: 돌 놓기

- 3×N 테이블의 각 칸에 양 또는 음의 정수가 기록되어 있다
- 조약돌을 놓는 방법 (제약조건)
  - 가로나 세로로 인접한 두 칸에 동시에 조약돌을 놓을 수 없다
  - 각 열에는 적어도 하나 이상의 조약돌을 놓는다
- 목표: 돌이 놓인 자리에 있는 수의 합을 최대가 되도록 조약돌 놓기

테이블의 예

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

합법적인 예

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

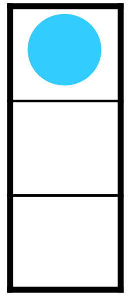
합법적이지 않은 예

6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

Violation!

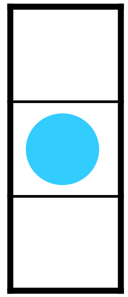
가능한 패턴

패턴 1:



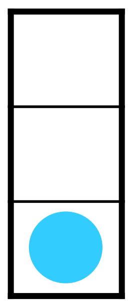
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

패턴 2:



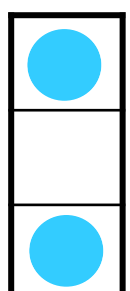
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

패턴 3:



6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

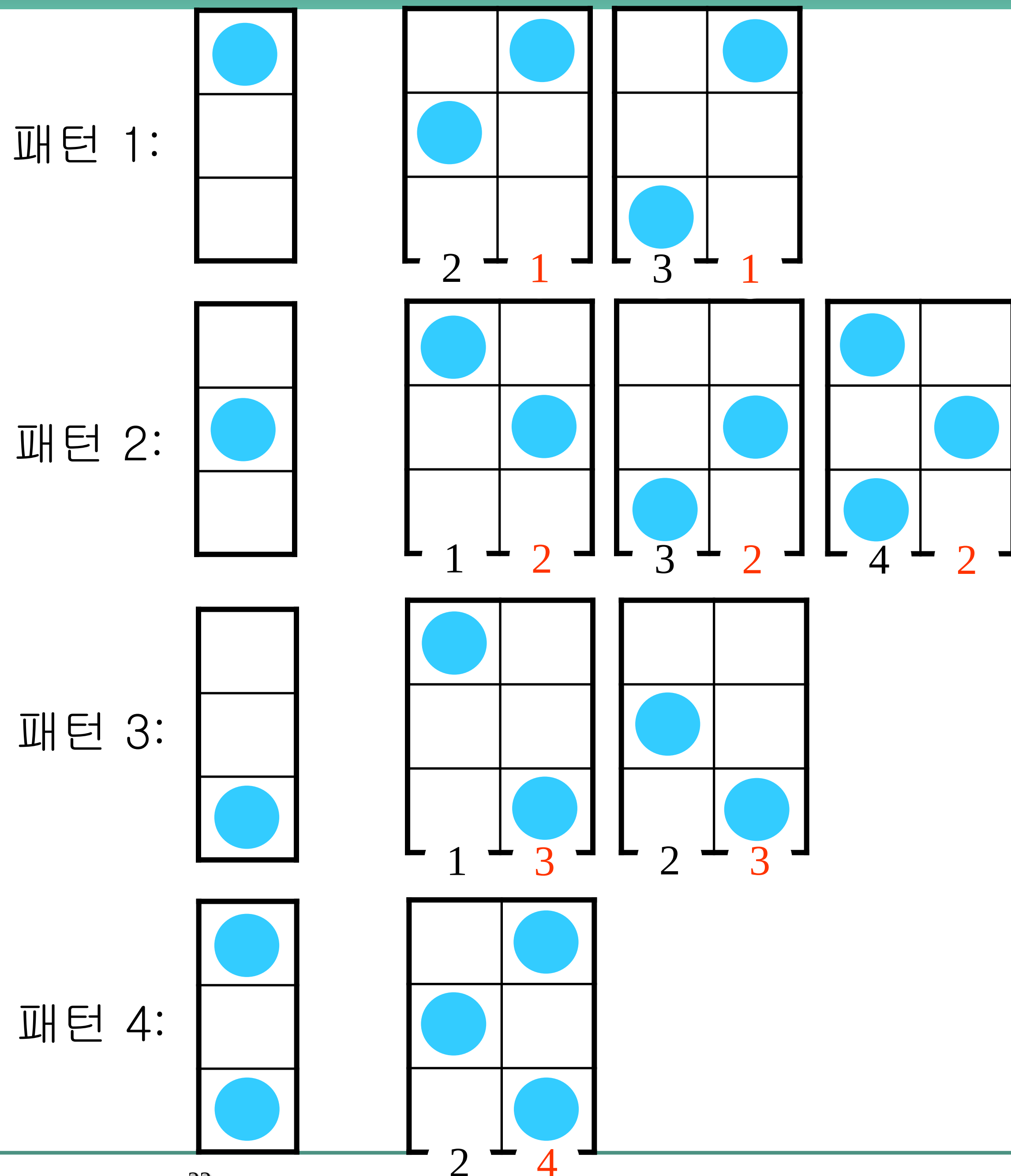
패턴 4:



6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

임의의 열을 채울 수 있는
패턴은 4가지뿐이다

서로 양립할 수 있는 패턴들



패턴 1은 패턴 2, 3과
패턴 2는 패턴 1, 3, 4와
패턴 3은 패턴 1, 2와
패턴 4는 패턴 2와 양립할 수 있다

i열과 i-1열의 관계

	i-1	i			
…	-5	5	3	11	3
	9	7	13	8	5
	4	8	-2	9	4

i-1열이 패턴 1로 끝나거나

i-1열이 패턴 3으로 끝나거나

i-1열이 패턴 4로 끝나거나

w[i, p]: i 열이 패턴 p로 놓일 때 i 열에 돌이 놓인 곳의 점수 합
peb[i, p]: i 열이 패턴 p로 놓을 때, 1-i 열의 최고 점수

peb[i, p]의 점화식

$$
\text{peb}[i, p] = \begin{cases} \text{w}[1, p] & \text{if i = 1} \\ \max\limits_{p\text{와 양립하는 패턴 } q}\{\text{peb}[i-1, q]\} + \text{w}[i, p] & \text{if i > 1} \end{cases}
$$

최종적으로 peb[n, 1] – peb[n, 4] 중 가장 큰 것이 답이다.

재귀 알고리즘

```
pebble(i, p)
▷ i 열이 패턴 p로 놓일 때의 i 열까지의 최대 점수 합 구하기
▷ w[i, p] : i 열이 패턴 p로 놓일 때 i 열에 돌이 놓인 곳의 점수 합. p∈{1, 2, 3, 4}
{
    if (i = 1)
            then return w[1, p] ;
            else {
                    max ← -∞ ;
                    for q ← 1 to 4 {
                            if (패턴 q가 패턴 p와 양립)
                            then {
                                    tmp ← pebble(i—1, q) ;
                                    if (tmp > max) then max ← tmp ;
                            }
                    }
                    return (max + w[i, p]) ;
            }
}
```

pebbleSum(n)

▷ n 열까지 조약돌을 놓은 방법 중 최대 점수 합 구하기

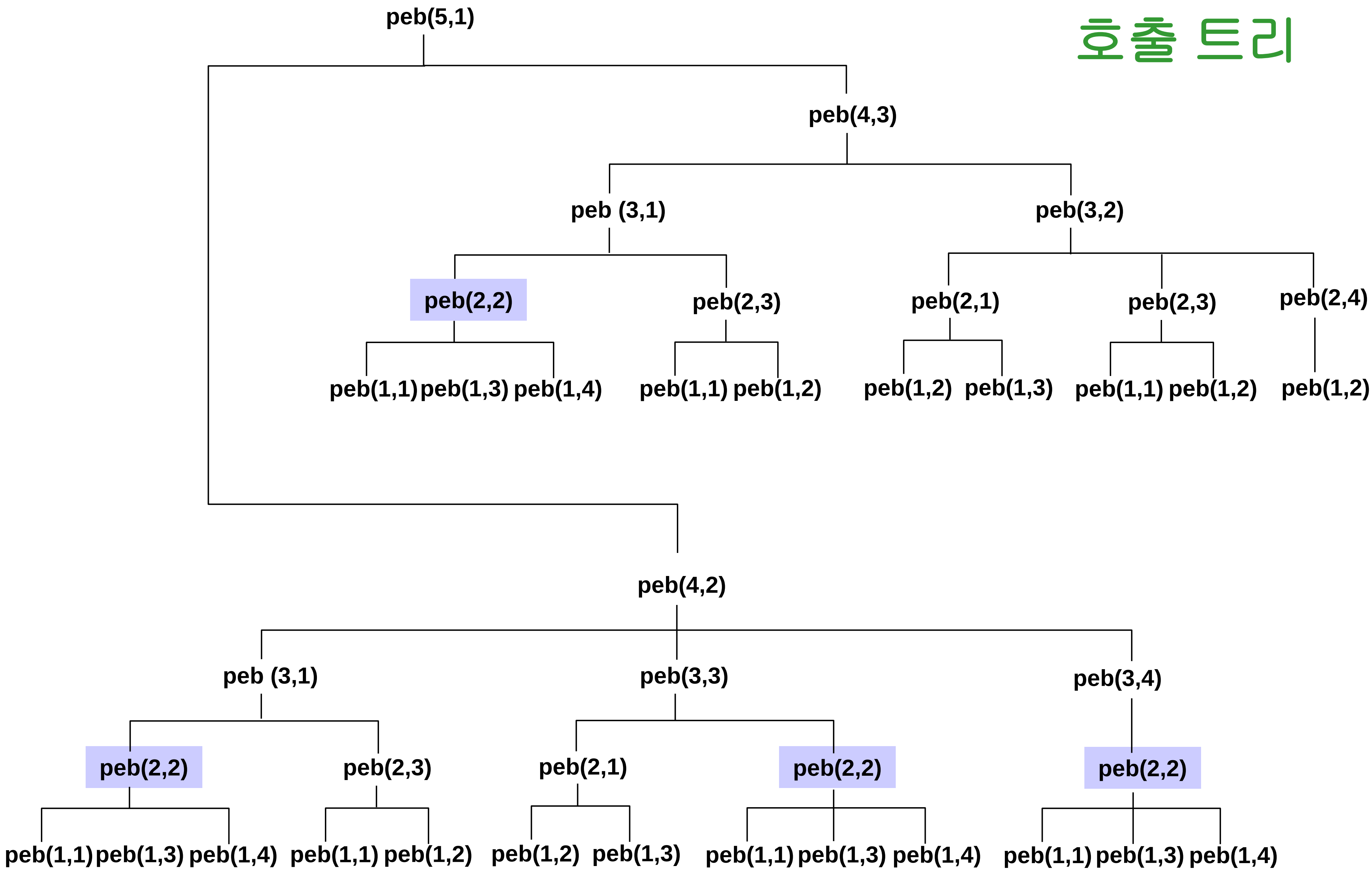
{

$$\textbf{return} \max_{p=1,2,3,4} \{ \text{pebble}(n, p) \} ;$$

}

✓ pebble(i, 1), …, pebble(i, 4) 중 최대값이 최종적인 답

호출 트리


peb(5,1)
peb(4,3)
peb (3,1)
peb(3,2)
peb(2,2)
peb(2,3)
peb(2,1)
peb(2,3)
peb(2,4)
peb(1,1) peb(1,3) peb(1,4)
peb(1,1) peb(1,2)
peb(1,2) peb(1,3)
peb(1,1) peb(1,2)
peb(1,2)
peb(4,2)
peb (3,1)
peb(3,3)
peb(3,4)
peb(2,2)
peb(2,3)
peb(2,1)
peb(2,2)
peb(2,2)
peb(1,1) peb(1,3) peb(1,4)
peb(1,1) peb(1,2)
peb(1,2) peb(1,3)
peb(1,1) peb(1,3) peb(1,4)
peb(1,1) peb(1,3) peb(1,4)

DP 적용

- DP의 요건 만족
  - 최적 부분구조
    - peb[*i*, .]에 peb[*i*-1, .]이 포함됨
    - 즉, 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함됨
  - 재귀호출시 중복
    - 재귀적 알고리즘에 중복 호출 심함

DP 알고리즘

```
pebble (n)
{
        for p ← 1 to 4
                peb[1, p] ← w[1, p] ;
        for i ← 2 to n
                for p ← 1 to 4
                        peb[i, p] ← max {peb[i-1, q]} + w[i, p] ;
                                   p와 양립하는 패턴 q
        return  max { peb[n, p] } ;
              p =1,2,3,4
}
```

✓복잡도 : $\Theta(n)$

복잡도 분석

기껏 4 바퀴

무시

기껏 n 바퀴

pebble(n)
{
for $p \leftarrow 1$ **to** 4
    peb[1, p] $\leftarrow$ w[1, p] ;
for $i \leftarrow 2$ **to** n
    for $p \leftarrow 1$ **to** 4
        peb[i, p] $\leftarrow$ max {peb[i-1, q]}+w[i, p] ;

p와 양립하는 패턴 q

return $\max_{p=1,2,3,4}$ { peb[n, p] } ;
}

기껏 3 가지

✓복잡도 : $\Theta(n)$

$n * 4 * 3 = \Theta(n)$

문제 예 3: 행렬 곱셈 순서

- 행렬 A, B, C
  - (AB)C = A(BC)
- 예: A:10 x 100, B:100 x 5, C:5 x 50
  - (AB)C: 10x100x5 + 10x5x50 = 7,500번의 곱셈 필요
  - A(BC): 100x5x50 + 10x100x50 = 75,000번의 곱셈 필요
- A_1, A_2, A_3, …, A_n을 곱하는 최적의 순서는?
  - 총 n-1회의 행렬 곱셈을 어떤 순서로 할 것인가?

재귀적 관계

- 마지막 행렬 곱셈이 수행되는 상황
  - n-1가지 가능성
    - $A_1(A_2 \dots A_n)$
    - $(A_1A_2)(A_3 \dots A_n)$
    - $(A_1A_2A_3)(A_4 \dots A_n)$
    - . . .
    - $(A_1 \dots A_{n-2})(A_{n-1}A_n)$
    - $(A_1 \dots A_{n-1})A_n$
  - 어느 경우가 가장 매력적인가?

- ✓ A_k의 차원: $p_{k-1}p_k$

- ✓ $m[i, j]$: 행렬 $A_i, \ldots, A_j$의 곱 $A_i \ldots A_j$를 계산하는 최소 비용

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

일반형: $(A_1 \ldots A_k)\ (A_{k+1} \ldots A_n)$

재귀적 구현

```
rMatrixChain(i, j)
▷ 행렬곱 A_i…A_j를 구하는 최소 비용 구하기
{
    if (i = j) then return 0;    ▷ 행렬이 하나뿐인 경우의 비용은 0
    min ← ∞;
    for k ← i to j-1 {
            q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + p_{i-1}p_kp_j;
            if (q < min) then min ← q;
    }
    return min;
}
```

✓ 엄청난 중복 호출이 발생한다!

동적 프로그래밍

```
matrixChain(i, j)
{
        for i ← 1 to n
                m[i, i] ← 0;    ▷ 행렬이 하나뿐인 경우의 비용은 0
        for r ← 1 to n-1         ▷ r: 문제 크기를 결정하는 변수, 문제의 크기 = r+1
                for i ← 1 to n-r {
                        j ← i+r;
                        m[i, j] ← ∞;
                        for k ← i to j-1 {
                                q ← min{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j};
                                if (q < m[i, j]) then m[i, j] ← q;
                        }
                }
        return m[1, n];
}
```

✓ 복잡도: $\Theta(n^3)$

$A_1, A_2, A_3, \ldots, A_n$을 곱하는 최적의 순서는?

```
matrixChain(i, j)
{
        for i ← 1 to n
                m[i, i] ← 0;     ▷ 행렬이 하나뿐인 경우의 비용은 0
        for r ← 1 to n-1          ▷ r: 문제 크기를 결정하는 변수, 문제의 크기 = r+1
                for i ← 1 to n-r {
                        j ← i+r;
                        m[i, j] ← ∞;
                        for k ← i to j-1 {
                                q ← min{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j};
                                if (q < m[i, j]) then {m[i, j] ← q; s[i, j] ← k;}
                        }
                }
        return m[1, n];
}
```

s[*i*, *j*]: 최소 비용 m[*i*, *j*]를 얻는 인덱스 *k*

문제 예 4: 최장 공통 부분순서LCS

- 두 문자열에 공통적으로 들어있는 공통 부분순서 중 가장 긴 것을 찾는다
- 부분순서의 예
  - <bcdb>는 문자열 <abcbdab>의 부분순서다
- 공통 부분순서의 예
  - <bca>는 문자열 <abcbdab>와 <bdcaba>의 공통 부분순서다
- 최장 공통 부분순서longest common subsequence(LCS)
  - 공통 부분순서들 중 가장 긴 것
  - 예: <bcba>는 문자열 <abcbdab>와 <bdcaba>의 최장 공통 부분순서다

최적 부분구조

- 두 문자열 $X_m = <x_1x_2 \ldots x_m>$과 $Y_n = <y_1y_2 \ldots y_n>$에 대해
  - $x_m = y_n$이면
    X_m과 Y_n의 LCS의 길이는 $X_{m\text{-}1}$과 $Y_{n\text{-}1}$의 LCS의 길이보다 1이 크다
  - $x_m \neq y_n$이면
    X_m과 Y_n의 LCS의 길이는
    X_m과 $Y_{n\text{-}1}$의 LCS의 길이와 $X_{m\text{-}1}$과 Y_n의 LCS의 길이 중 큰 것과 같다

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

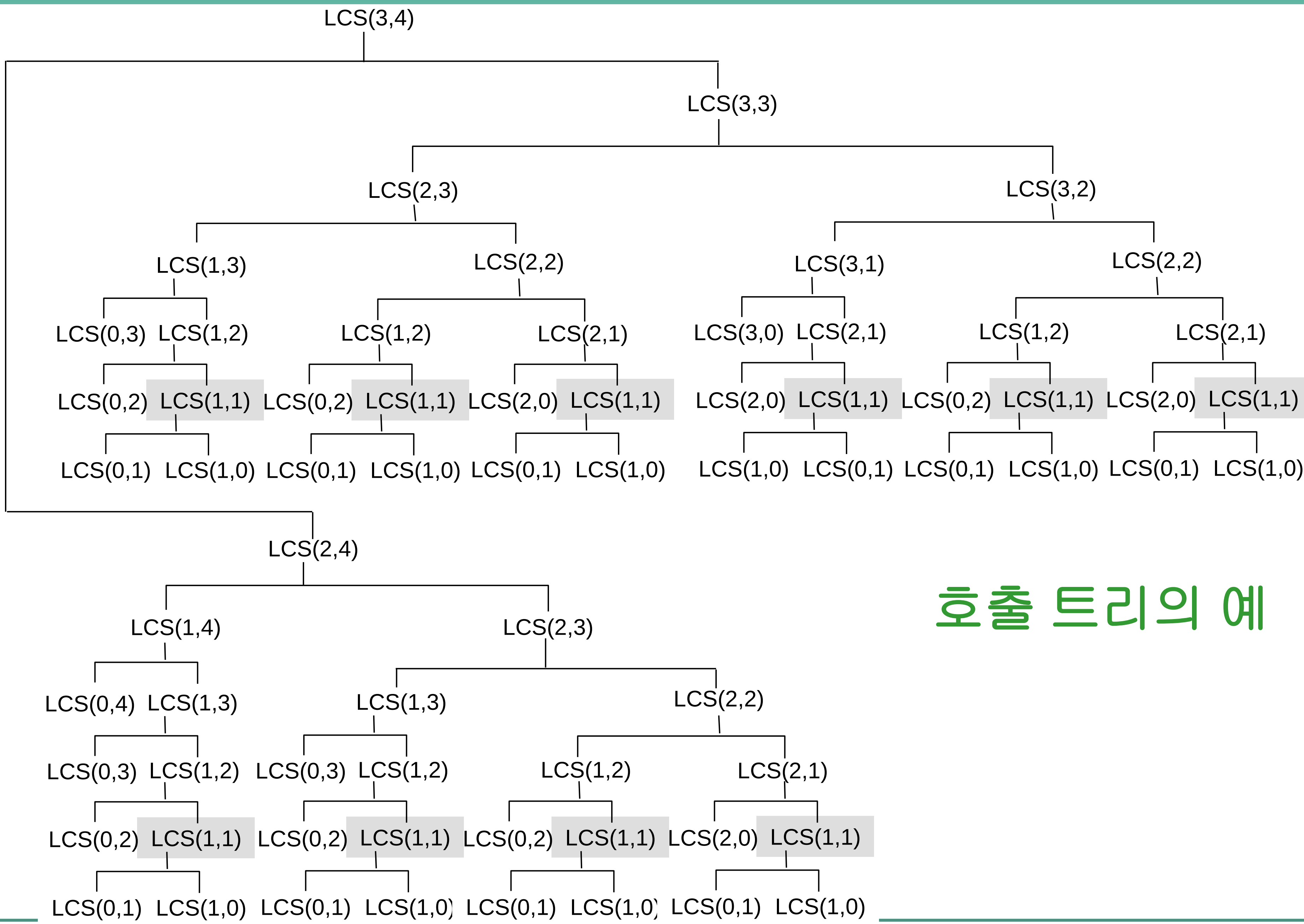
✓ $C[i, j]$: 두 문자열 $X_i = <x_1x_2 \ldots x_i>$과 $Y_j = <y_1y_2 \ldots y_j>$의 LCS 길이

재귀적 구현

```
LCS(m, n)
▷ 두 문자열 X_m과 Y_n의 LCS 길이 구하기
{
    if (m = 0 or n = 0) then return 0;
    else if (x_m = y_n) then return LCS(m-1, n-1) + 1;
    else return max(LCS(m-1, n), LCS(m, n-1));
}
```

✓ 엄청난 중복 호출이 발생한다!

호출 트리의 예


LCS(3,4)
LCS(3,3)
LCS(2,3) LCS(3,2)
LCS(1,3) LCS(2,2) LCS(3,1) LCS(2,2)
LCS(0,3) LCS(1,2) LCS(1,2) LCS(2,1) LCS(3,0) LCS(2,1) LCS(1,2) LCS(2,1)
LCS(0,2) LCS(1,1) LCS(0,2) LCS(1,1) LCS(2,0) LCS(1,1) LCS(2,0) LCS(1,1) LCS(0,2) LCS(1,1) LCS(2,0) LCS(1,1)
LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0) LCS(1,0) LCS(0,1) LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0)
LCS(2,4)
LCS(1,4) LCS(2,3)
LCS(0,4) LCS(1,3) LCS(1,3) LCS(2,2)
LCS(0,3) LCS(1,2) LCS(0,3) LCS(1,2) LCS(1,2) LCS(2,1)
LCS(0,2) LCS(1,1) LCS(0,2) LCS(1,1) LCS(0,2) LCS(1,1) LCS(2,0) LCS(1,1)
LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0) LCS(0,1) LCS(1,0)

동적 프로그래밍

LCS(m, n)
▷ 두 문자열 X_m과 Y_n의 LCS 길이 구하기
{
    for $i \leftarrow 0$ **to** m
        C[i, 0] $\leftarrow$ 0;
    for $j \leftarrow 0$ **to** n
        C[0, j] $\leftarrow$ 0;
    for $i \leftarrow 1$ **to** m
        for $j \leftarrow 1$ **to** n
            if ($x_i = y_j$) **then** C[i, j] $\leftarrow$ C[i-1, j-1] + 1;
            else C[i, j] $\leftarrow$ max(C[i-1, j], C[i, j-1]);
    return C[m, n];
}

✓ 복잡도: $\Theta(mn)$

Thank you