2021-2
알고리즘

동적 프로그래밍

한남대학교 컴퓨터공학과

Review: 재귀 호출의 실용성

- **현실적인 문제**
  - 메모리의 스택 영역에 스택 프레임이 최소 n개 쌓인다: **Stack Overflow!**
  - 잘못 구현하면 시간, 공간 효율성이 극단적으로 나빠짐

- 그럼에도
  - 컴퓨터 과학의 근간을 이루는 개념 중 하나이며,
  - 함수형 프로그래밍(Functional Programming)에서 재귀 호출은 시작이자 끝이라고 할 수 있다.
  - "재귀"의 개념 없이는 풀 수 없는 문제들도 많다.

- **재귀 호출을 보완하는 기법들**
  - **Memoization, 꼬리 재귀, 가지치기** 등
  - 재귀 호출 알고리즘을 반복문으로 구현 가능

8장 구성

- **재귀적 해법의 문제점**

- **동적 프로그래밍**
- **Memoization**

- **DP 문제 예시**
  - 계단 밟기 문제
  - 행렬 경로 문제
  - 최장 부분 공통순서LCS 문제
  - 조약돌 놓기 문제

재귀적 해법의 문제점

배경

- **재귀적 해법recursive solution**
  - 큰 문제에 닮음꼴의 작은 문제가 깃든다.
  - 관계중심으로 파악함으로써 문제를 간명하게 볼 수 있다.

- **문제점**
  - 1. 많은 메모리 공간 사용: Stack Overflow 발생
  - 2. 중복 호출 duplicated function calls

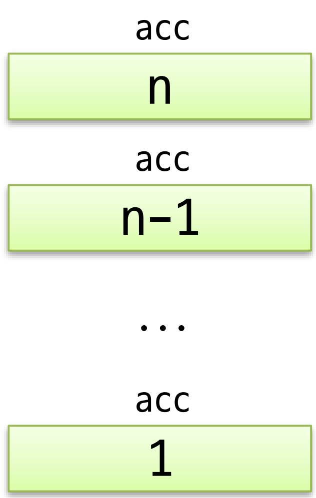
- **원인**
  - **Mathmatical Logic**과 **Computing Logic** 사이의 괴리

재귀적 해법의 문제점

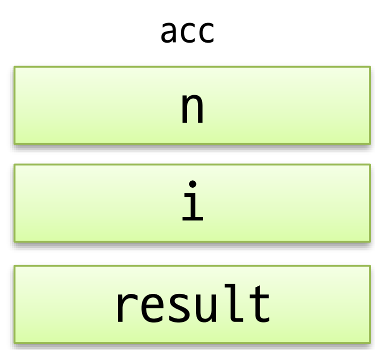
- **문제점 1: Stack Overflow**
  - 스택 프레임의 개수는 유한하다.
  - 반복문으로 n번 반복하는 시간적, 공간적 비용 <<< 함수를 n번 호출하는 비용

```
def acc(n):
    if n == 1:
        return 1
    else:
        return n + acc(n – 1)
```

```
def acc(n):
    result = 0
    for i in range(1, n+1):
        result += i

    return result
```



n의 크기만큼
stack frame을 할당



i = 1 to n

메모리 사용량은 n의 크기와 무관함

꼬리 재귀

- **꼬리 재귀(Tail Recursion)**
  - **함수의 끝에서 재귀 호출**을 했을 때,
  - 컴파일 과정에서 반복문으로 변환해주는 기능이 존재
  - 스택 프레임이 쌓이는 문제를 보완해주지만 **컴파일러에 따라 이 기능이 있을 수도, 없을 수도 있음**

```
def acc(n):
    if n == 1:
        return 1
    else:
        return n + acc(n – 1)
```

꼬리 재귀

- **꼬리 재귀(Tail Recursion)가 가능한 코드**
  - 반복문을 재귀 호출로 구현한 경우

```
yes_or_no() {
    while True {
        answer <- 키보드에서 소문자 한 개를 입력받음;
        answer가 'y' 또는'n'이면 return answer;
    }
}
```

```
yes_or_no() {
    answer <- 키보드에서 소문자 한 개를 입력받음;
    answer가 'y' 또는'n'이면 return answer;
    yes_or_no();
}
```

꼬리 재귀

- **꼬리 재귀(Tail Recursion)가 가능한 코드**
  - Binary Search Algorithm

```
def binary_search(sorted_arr, p, r, target):
    if p > r:
        return False

    q = (p + r) // 2
    if sorted_arr[q] == target:
        return True
    elif target < sorted_arr[q]:
        r = q - 1   # left
    else:
        p = q + 1   # right

    return binary_search(sorted_arr, p, r, target)
```

재귀적 해법의 문제점

- **문제점2: 중복 호출**duplicated function calls

- **도입문제: 피보나치 수fibonacci number 구하기**
  - $f(n) = f(n-1) + f(n-2)$
  - $f(1) = f(2) = 1$

  - 아주 간단한 문제지만 동적 프로그래밍의 동기와 구현이 다 포함되어 있다.

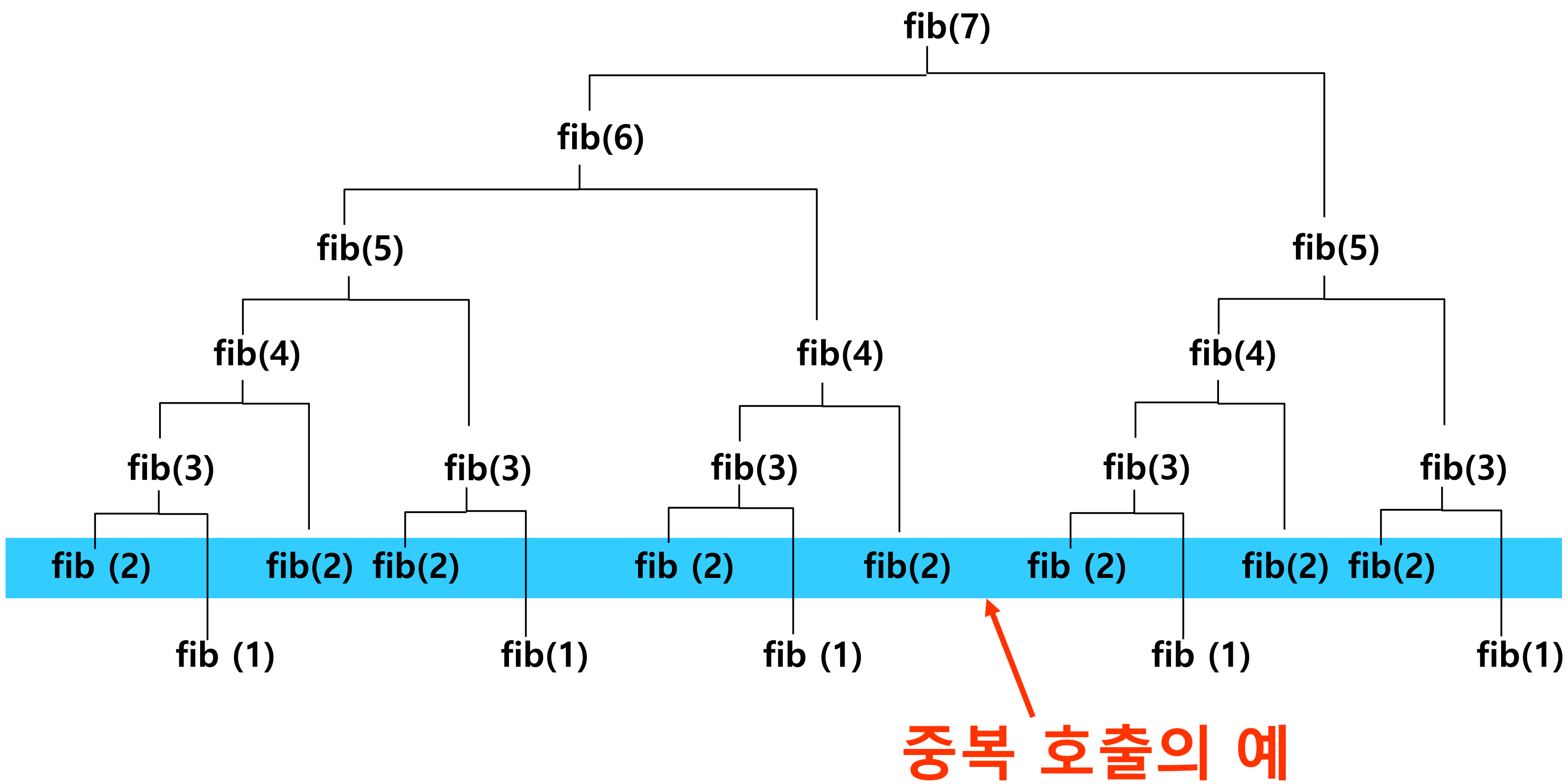
재귀적 해법의 문제점

- **문제점2: 중복 호출**duplicated function calls

```
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



재귀적 해법의 문제점

- **재귀적 해법이 바람직한 예**
  - 퀵정렬, 병합정렬 등의 정렬 알고리즘
  - 계승(factorial) 구하기
  - 그래프의 DFS
  - …
- **엄청난 중복 호출이 발생하는 예**
  - 피보나치수 구하기
  - 행렬곱셈 최적순서 구하기
  - …

동적 프로그래밍

동적 프로그래밍의 적용 요건

- **최적 부분구조optimal substructure**
  – 큰 문제의 최적해에 작은 문제의 최적해가 포함됨

- **재귀호출 시 중복overlapping recursive calls**
  – 재귀적 해법으로 풀면 같은 문제에 대한 재귀호출이 심하게 중복됨

⇨ 동적 프로그래밍이 그 해결책!

동적 프로그래밍

- **동적 프로그래밍**
- **(동적 계획법, Dynamic Programming, DP)**
  - 재귀적 해법으로 풀면 (크기가) 같은 문제에 대한 재귀호출이
  - 심하게 중복되는 문제
- → 재귀적 해법에서
  - **한번 풀이한 문제의 해solution을 기억**해서
  - 중복 호출을 제거

피보나치수를 구하는 동적 프로그래밍 알고리즘

```
fibonacci(n)
{
      f[1] ← f[2] ← 1;
      for i ← 3 to n          3번째 수, 4번째 수, ⋯, n번째 수를 차례로 구한다.
            f[i] ← f[i-1] +f[i-2];
      return f[n];
}
```

✓ 선형시간에 끝난다

재귀적 해법이지만, 재귀 호출이 아니라
sub-optimal solution을 **기억할 배열**과 **반복문**으로 구성

연습문제

- **피보나치수를 구하는 앞의 두 함수의 실행 시간을 대략 비교해 보자.**
  - python이라면 우선 재귀 호출 깊이의 한계를 늘려 준다.

```
import sys

sys.setrecursionlimit(5000)
```

적당히 큰 수

예제

- **1부터 n까지 합을 구하는 프로그램에 동적 프로그래밍을 적용**
  - 중복 호출이 일어나지 않으므로 dp가 꼭 필요한 문제는 아님. 연습을 위해 사용함

```
def acc(n):
    if n == 1:
        return 1
    else:
        return n + acc(n – 1)
```

```
def acc(n):
    # 배열과 초깃값 준비
    sol = [0 for _ in range(n+1)]
    sol[1] = 1

    for i in range(2, n+1):   # 2부터 n까지 해를 구한다.
        sol[i] = i + sol[i-1]

    return sol[n]
```

동적 프로그래밍을 어려워하는 이유

- **DP는 재귀적인 해법을 구하고 → 동적 프로그래밍으로 구현**
  - 하지만 대부분 학생들은 이 단계를 건너 뛰고 DP 알고리즘을 바로 작성하려고 시도한다.
- **DP가 어려운 가장 큰 이유는 재귀적인 사고 훈련이 덜 되어 있기 때문**
  - DP는 새로운 방법이 아니라 재귀적인 해법의 다른 표현일 뿐이다.

동적 프로그래밍을 어려워하는 이유

- **DP와 재귀호출의 구현 상 차이점**
  - 해를 구하는 순서

- **재귀 호출**
  - **더 작은 해를 구했다고 가정**하고 더 큰 해를 구한다.
  - 실제로 호출 순서는 n → n-1 → n-2 … → 2 → 1

```
return n + acc(n – 1)
```

- **DP**
  - **더 작은 해를 먼저 구해 둔 후에** 더 큰 해를 구한다.
  - 반복문이 실행되는 순서는 1 → 2 → … → n-1 → n

```
for i in range(2, n+1):
    sol[i] = i + sol[i-1]
```

동적 프로그래밍을 어려워하는 이유

- **DP가 어렵다면**
- **재귀 문제를 만났을 때 3가지 버전으로 작성하는 연습을 추천**
  - 주어진 문제를 재귀 호출로 구현(완전 탐색 알고리즘)
  - → 바로 뒤에서 배울 Memoizaiton으로 수정
  - → 다시 처음부터 DP로 작성

MEMOIZATION

동적 프로그래밍

- **Memoization(메모이제이션, 메모아이제이션)**
  - 한번 구한 해를 기억한다.
  - DP의 핵심 기법이며, DP의 일종으로 볼 수도 있다.

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

  - 우선 **재귀 호출로 먼저 구현**한 후,
  - Memoization을 적용해서 중복 호출을 제거한다.

동적 프로그래밍

- **Memoization(메모이제이션, 메모아이제이션)**
  - 한번 구했던 해를 기억해서 재활용한다.

```
def fib(n):
    global dp  # 함수의 인자로 넘겨주는 게 더 일반적임

    if n in dp:                    2) 기억해둔 해가 있으면 사용
        return dp[n]

    dp[n] = fib(n-1) + fib(n-2)    3) return하기 전에
    return dp[n]                      새로 구한 해를 dp에 기억시킴


dp = {1:1, 2:1}                    1) sub-optimal solution들을 기억할 자료구조
print(fib(6))
```

연습문제

- **앞에서 1부터 n까지 합을 구한 프로그램이다.**
  - Memoization을 적용해 보자.
  - **중복 호출이 일어나지 않으므로, dp가 꼭 필요한 문제는 아님

```
def acc(n):
    if n == 1:
        return 1
    else:
        return n + acc(n - 1)
```

  - 1) 해를 리턴하기 전에 기억시킨다.
  - 2) 기억해둔 해가 있으면 재귀 호출하지 않고 기억시킨 해를 사용한다.

연습문제

- **앞에서 작성한 코드에서 recursion depth가 1 줄어든 코드를 작성해 보자.**
  - 예를 들어, 아래 코드에서
    - fib(3)에서 fib(2), fib(1)을 호출하면 바로 dp[2], dp[1]을 리턴하지 않고
    - fib(2)이 호출된 후에 dp[2]를 리턴
    - fib(1)이 호출된 후에 dp[1]을 리턴한다.

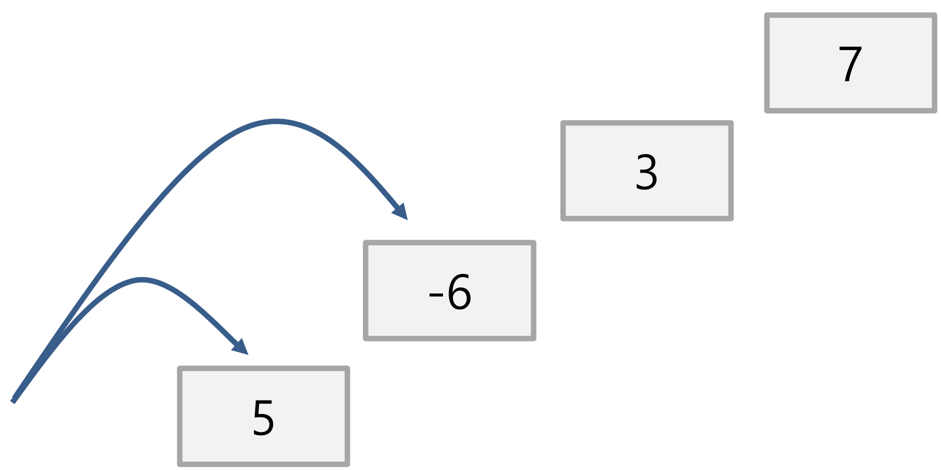
```
def fib(n):
    global dp

    if n in dp:
        return dp[n]

    dp[n] = fib(n-1) + fib(n-2)
    return dp[n]


dp = {1:1, 2:1}
print(fib(6))
```

계단 밟기 문제

계단 밟기 문제

- **계단을 1칸 또는 2칸을 오를 수 있다.**
- **입력**
  - 계단마다 얻을 수 있는 점수가 있고, 계단을 밟으면 그 계단의 점수를 더한다.
  - 점수는 음수가 될 수도 있다. 계단의 개수 n과 계단의 점수들이 입력으로 주어진다.
  - **$S = \{s_1, s_2, \cdots, s_n\}$**

- **출력**
  - 계단을 끝까지 올라서 얻을 수 있는 가장 많은 점수



계단 밟기 문제

- **문제의 정의: M(S, k)** (편의상 **M(k)**라고 하자)
  - k번째 계단까지 올라서 얻을 수 있는 **가장 많은 점수**

- **풀이: k번째 계단**을 오르는 방법은 두 가지
  - **(k-1)번째 계단에서** 한 칸 오르기
  - → $\underbrace{\text{(k-1)번째 계단까지 최대점수}}_{M(k-1)}$ + k번째 계단의 점수
  - **(k-2)번째 계단에서** 두 칸 오르기
  - → $\underbrace{\text{(k-2)번째 계단까지 최대점수}}_{M(k-2)}$ + k번째 계단의 점수

– $\mathbf{M(k) = s_1 + \max(M(k-1), M(k-2))}$

– $M(0) = 0, M(1) = s_1$

계단 밟기 문제

- **재귀적인 해법**

```
M(S, k) {
    if k <= 1 return S[k]

    return S[k] + max(M(S, k-1), M(S, k-2))
}
```

계단 밟기 문제

- **Memoization**

```
M(S, k) {
    if k <= 1 return S[k]

    return S[k] + max(M(S, k-1), M(S, k-2))
}
```

```
dp[0] = 0, dp[1] = S[1]로 초기화한다.

M(S, k, dp) {
    if k in dp then return dp[k]

    dp[k] = S[k] + max(M(S, k-1), M(S, k-2))
    return dp[k]
}
```

계단 밟기 문제

- **Dynamic Programming**

```
M(S, n) {
    dp[0] <- 0
    dp[1] <- S[1]

    for i in 2..n {
       dp[i] = S[i] + max(dp[k-1], dp[k-2])
    }

    return dp[n]
}
```

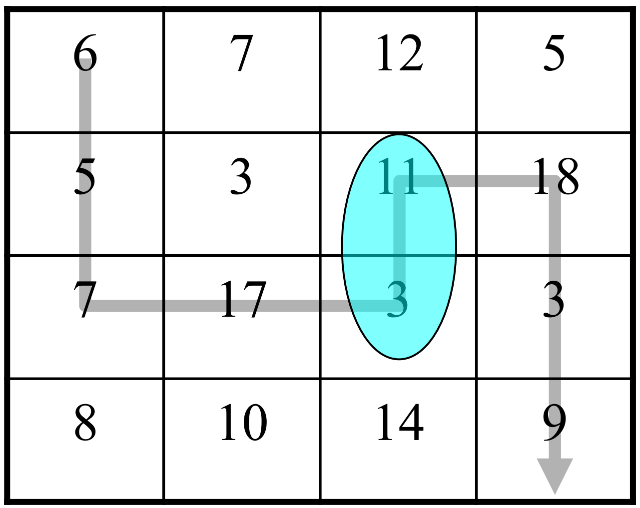
행렬 경로 문제

행렬 경로 문제

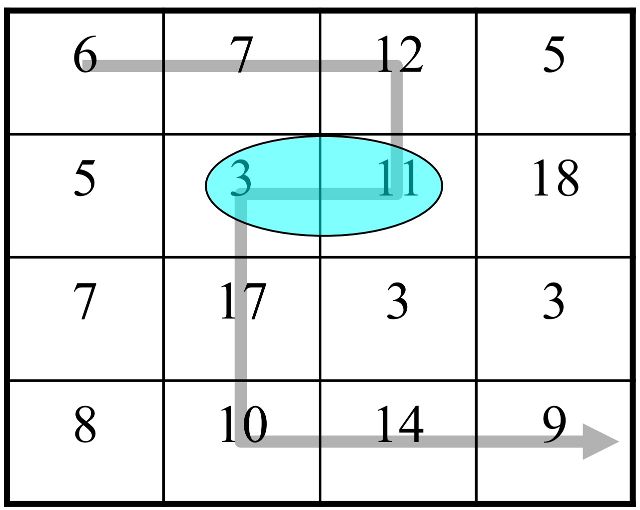
- **양수 원소들로 구성된 $n \times n$ 행렬이 주어지고, 행렬의 좌상단에서 시작하여 우하단까지 이동한다**
- **이동 방법 (제약조건)**
  - 오른쪽이나 아래쪽으로만 이동할 수 있다. → ↓
  - 왼쪽, 위쪽, 대각선 이동은 허용하지 않는다.

- **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

- **문제의 정의: matrixPath(*i, j*)**
  - (1, 1)에서 (i, j)까지 오는 모든 경로에 대해,
  - <하나의 경로에서 경로 상의 숫자들을 더한 값> 중의 최댓값

<경로합>이라고 부르자.

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

경로합 =
6 + 7 + 12 + 11 + 18 + 3 + 9

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

경로합 =
6 + 5 + 7 + 17 + 10 + 14 + 9

→ 가능한 모든 경로에서 합의 최댓값

재귀 알고리즘

- **문제의 정의: matrixPath(*i*, *j*)**
  - (1, 1)에서 (i, j)까지 오는 모든 경로에 대해,
  - <하나의 경로에서 경로 상의 숫자들을 더한 값> 중의 최댓값

- **재귀적인 해법**
  - 한 칸을 움직여서 (i, j)로 오는 방법은 두 가지: (i, j-1), (i-1, j)

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

(i-1, j)

(i, j)

(i, j-1)

재귀 알고리즘

- **재귀적인 해법**
  - (i, j)로 오는 경로는 두 개: (i-1, j), (i, j-1)
  - **mat(i-1, j)**: (1, 1)에서 (i-1, j)로 오는 모든 경로합 중 최댓값
  - **mat(i, j-1)**: (1, 1)에서 (i, j-1)로 오는 모든 경로합 중 최댓값

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

mat(i-1, j)

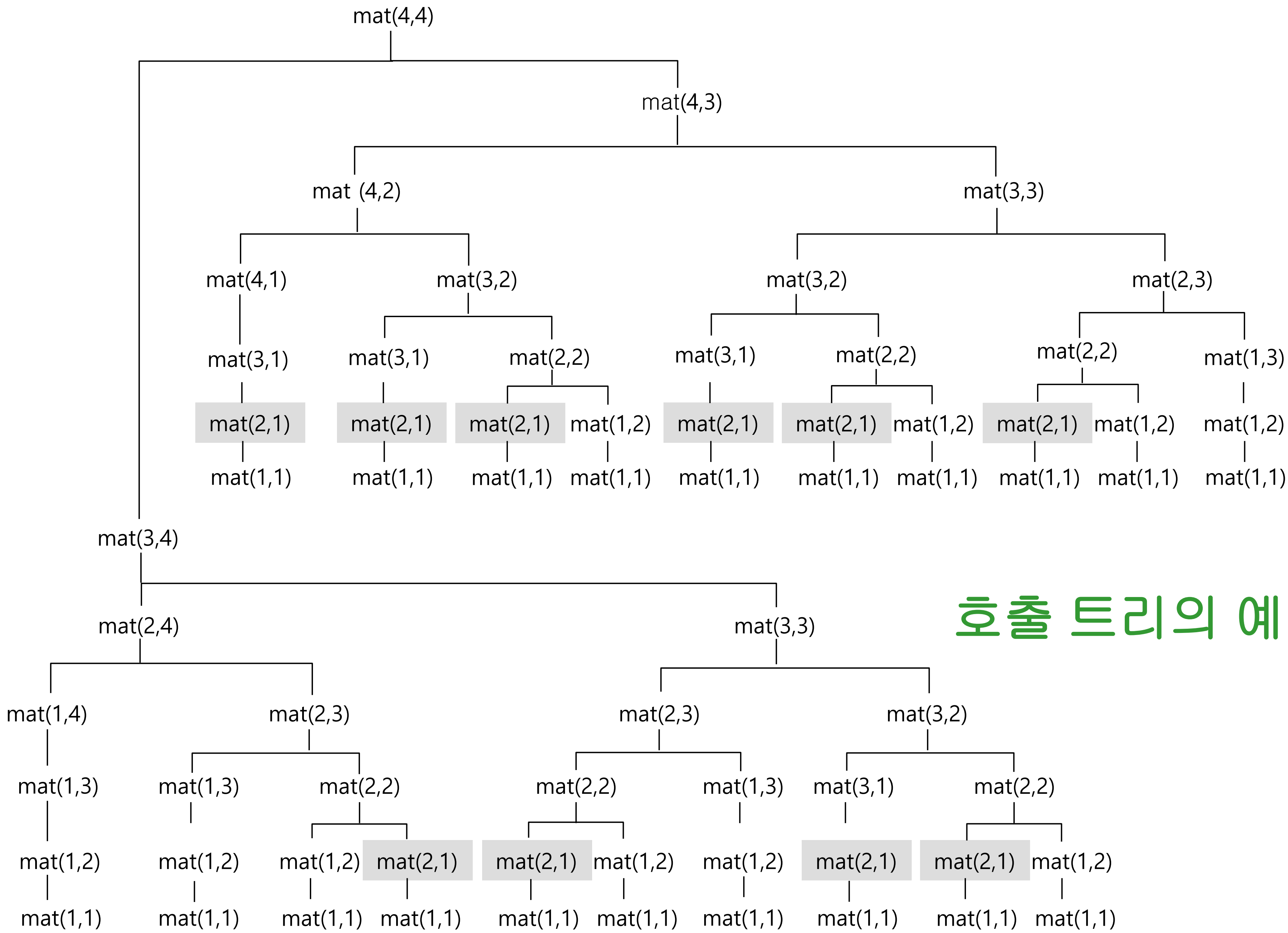
6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

mat(i, j-1)

  - → $\text{mat}(i, j) = m_{ij} + \max(\text{mat}(i-1, j), \text{mat}(i, j-1))$

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 알고리즘

- $mat(i, j) = m_{ij} + max(mat(i-1, j), mat(i, j-1))$

- → **mat(i, j)**를 구하기 전에 **mat(i-1, j), mat(i, j-1)**을 먼저 구해두면
- 그 값으로 **mat(i, j)**를 구할 수 있다.
  - 행렬을 linear scan하면 이 조건을 만족함

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

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];
}
```

최장 공통 부분순서 문제

최장 공통 부분순서 문제(LCS)

- LCS 문제
  - **두 문자열에 공통적으로 들어있는 공통 부분순서 중 가장 긴 것의 길이를 찾는다.**
  - **부분순서Subsequence** 의 예
    - <bcdb>는 문자열 <abcbdab>의 부분순서다.
    - (연속될 필요 없음. non-continuous)
  - **공통 부분순서Common Subsequence** 의 예
    - <bca>는 문자열 <abcbdab>와 <bdcaba>의 공통 부분순서다.
  - **최장 공통 부분순서Longest Common Subsequence (LCS)**
    - 공통 부분순서들 중 가장 긴 것(의 길이)
    - 예: <bcba>는 문자열 <abcbdab>와 <bdcaba>의 최장 공통 부분순서다.

최장 공통 부분순서 문제(LCS)

- **문제의 정의: 크기 (I, j)인 문제**
  - c_{ij} : 두 문자열 $X_i = <x_1x_2 \cdots x_i>$과 $Y_j = <y_1y_2 \cdots y_j>$의 LCS 길이

- **크기 (i-1, j)인 문제**
  - $c_{i-1,j}$: 두 문자열 $X_i = <x_1x_2 \cdots x_{i-1}>$과 $Y_j = <y_1y_2 \cdots y_j>$의 LCS 길이

- **크기 (i, j-1)인 문제**
  - $c_{i,j-1}$: 두 문자열 $X_i = <x_1x_2 \cdots x_i>$과 $Y_j = <y_1y_2 \cdots y_{j-1}>$의 LCS 길이

- **가정:**
  - $c_{i-1,j}$ 와 $c_{i,j-1}$ 가 주어짐

- **찾아야할 것:**
  - c_{ij} 와 $c_{i-1,j}$, 또는 c_{ij} 와 $c_{i,j-1}$ 의 관계

최적 부분구조

- **두 문자열 $X_m = \langle x_1 x_2 \cdots x_m \rangle$과 $Y_n = \langle y_1 y_2 \cdots y_n \rangle$에 대해**
- **마지막 문자를 비교**

- **$x_m = y_n$이면**

  X_m과 Y_n의 LCS의 길이는 X_{m-1}과 Y_{n-1}의 LCS의 길이보다 1이 크다.

$X_m = \cdots a$ $Y_n = \cdots a$

- **$x_m \neq y_n$이면**

  X_m과 Y_n의 LCS의 길이는

  X_m과 Y_{n-1}의 LCS의 길이와 X_{m-1}과 Y_n의 LCS의 길이 중 큰 것과 같다.

$X_m = \cdots b$ $Y_n = \cdots a$

최적 부분구조

✓ c_{ij} : 두 문자열 $X_i = <x_1 x_2 \ldots x_i>$과 $Y_j = <y_1 y_2 \ldots y_j>$의 LCS 길이

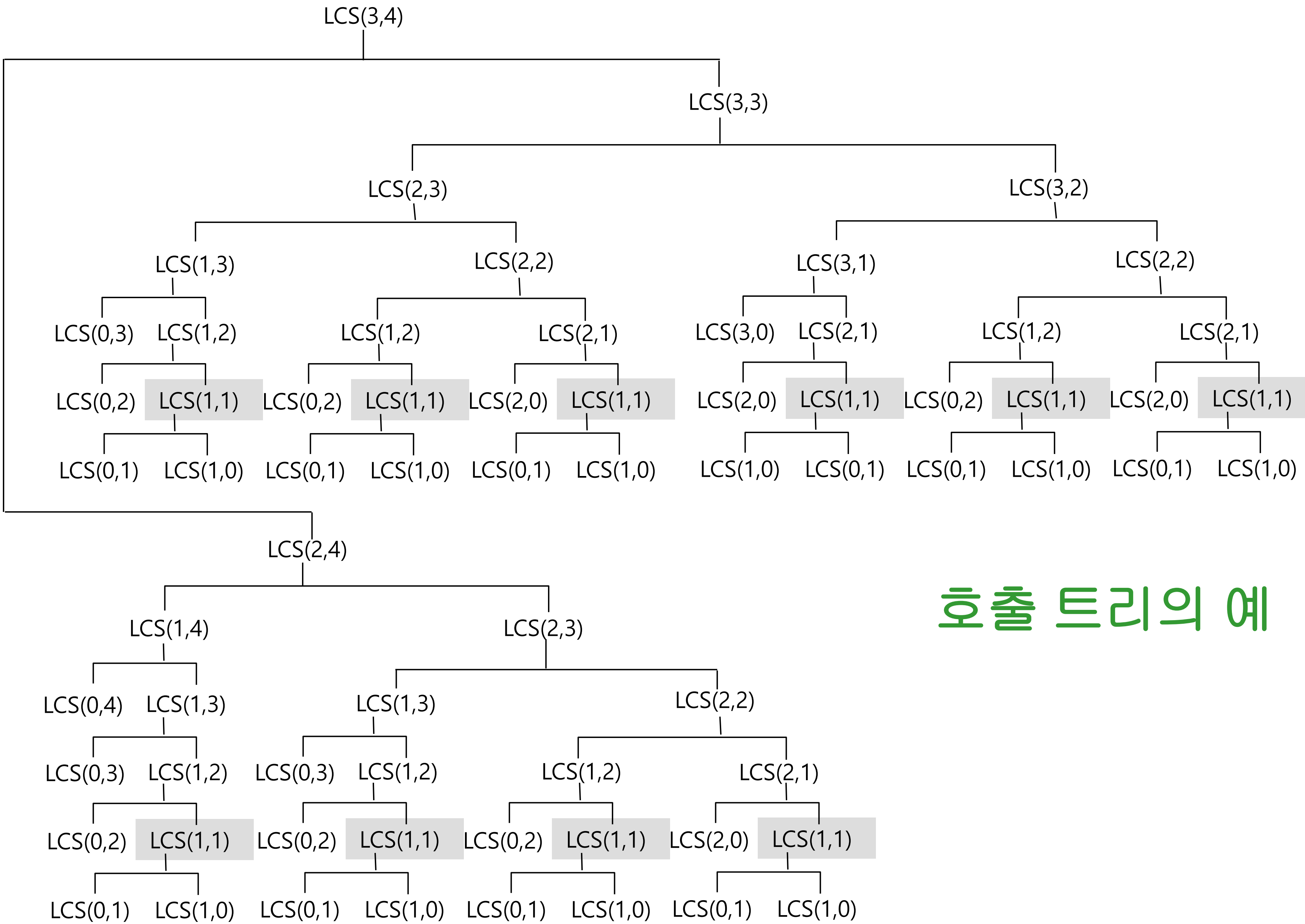
- $$c_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c_{i-1,\, j-1} + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c_{i-1,\, j},\ c_{i,\, j-1}\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

재귀적 구현

```
LCS(m, n)
▷ 두 문자열 Xm과 Yn의 LCS 길이 구하기
{
    if (m = 0 or n = 0) then   return 0;
    else if (xm = yn) 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 Xm과 Yn의 LCS 길이 구하기
{
    for i ← 0 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 (xi = yj) then C[i, j] ← C[i-1, j-1] + 1;
            else              C[i, j] ← max(C[i-1, j], C[i, j-1]);

  return C[m, n];
}
```

✓ 복잡도: $\Theta(mn)$

조약돌 놓기 문제

조약돌 놓기 문제

- 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

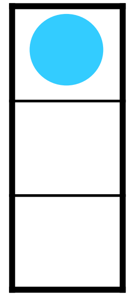
합법적이지 않은 예

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!

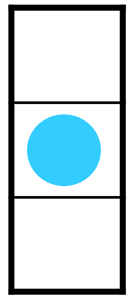
가능한 패턴

패턴 1:



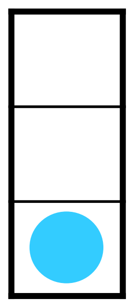
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

패턴 2:



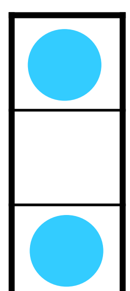
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

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

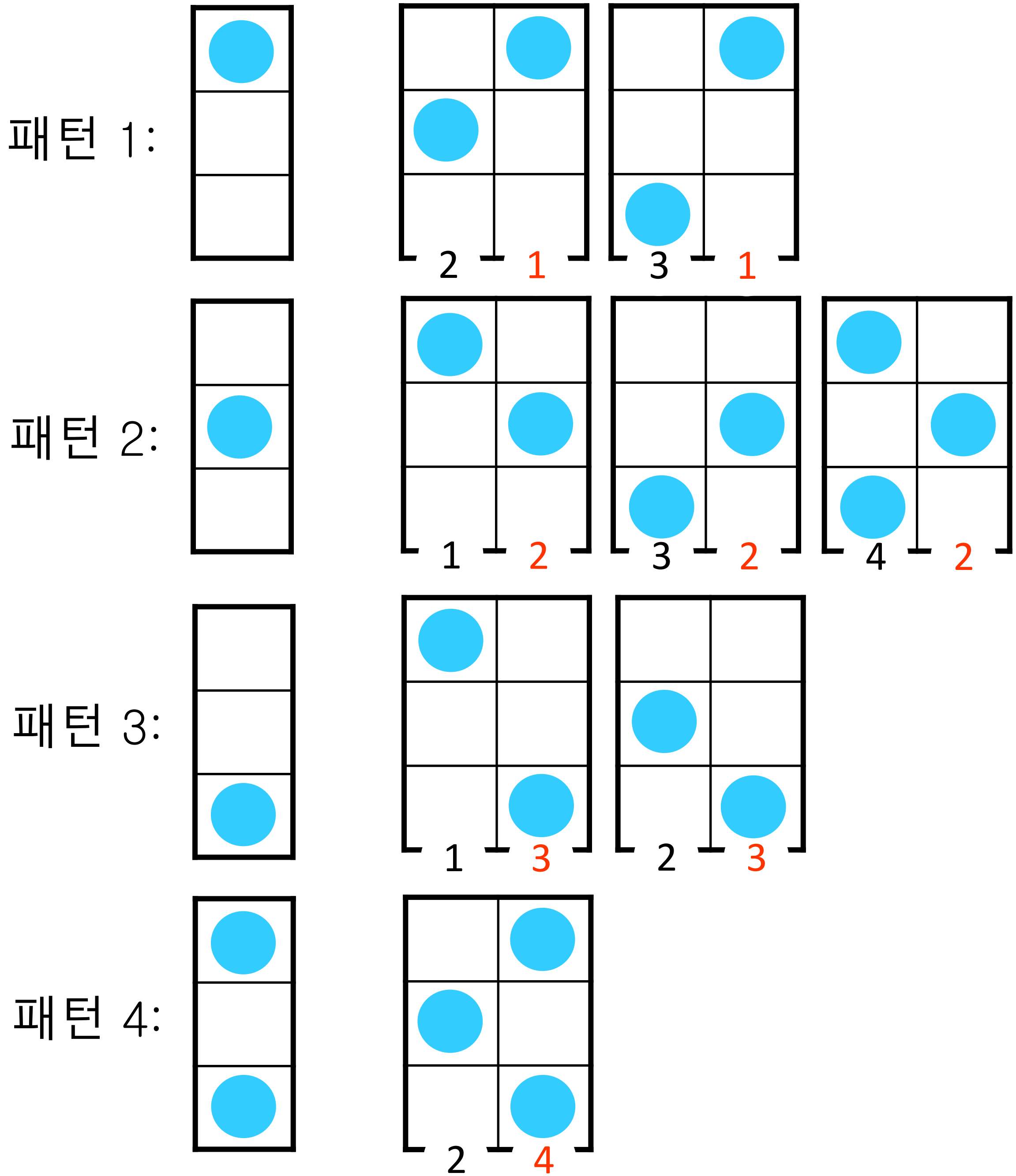
패턴 4:



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

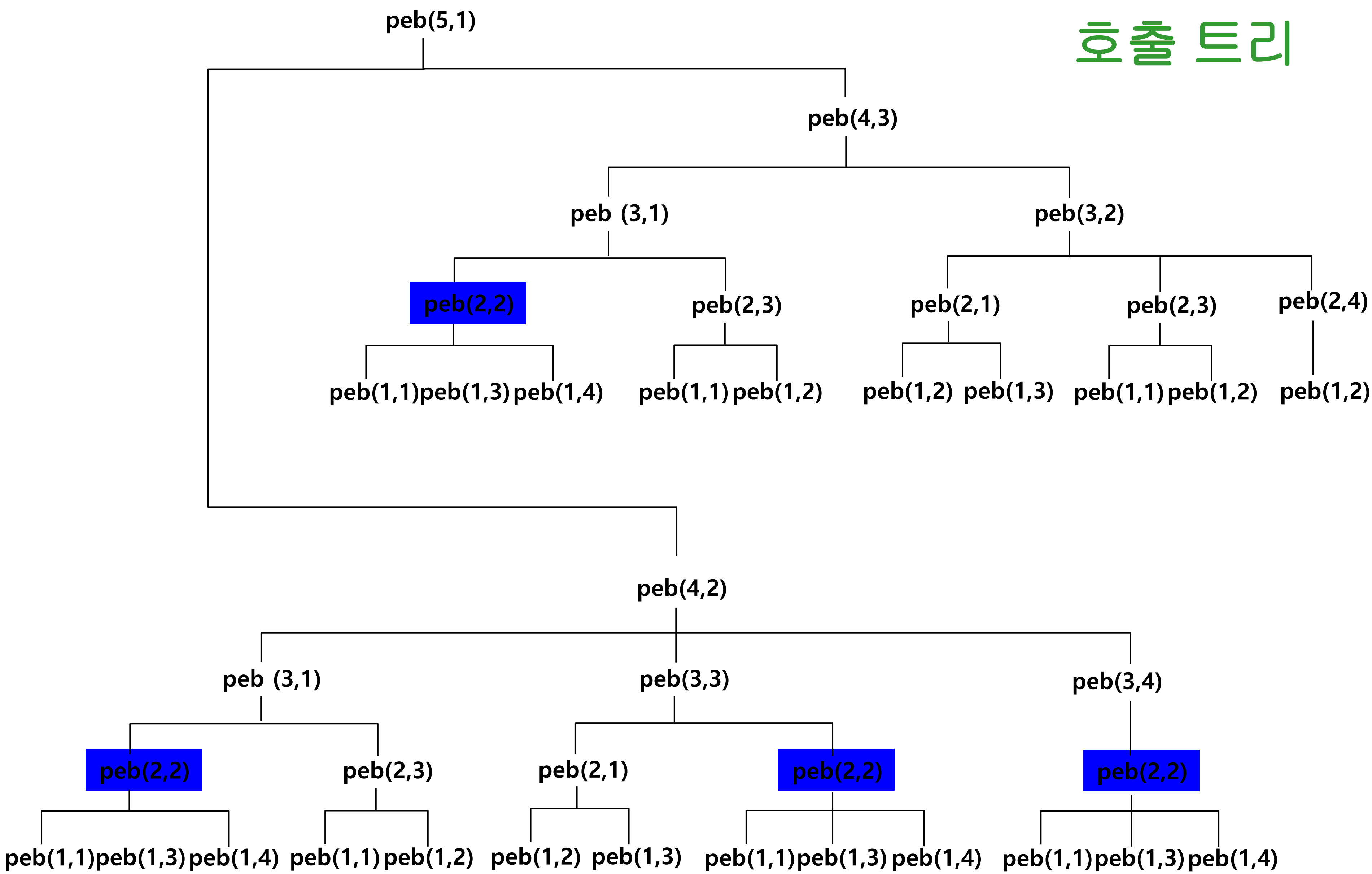
재귀 알고리즘

```
pebble(i, p)
▷ i열이 패턴 p로 놓일 때의 i열까지의 최대 점수 합 구하기
▷ w[i, p] : i열이 패턴 p로 놓일 때 i열에 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
{
return max { pebble(*n*, *p*) } ;
p =1,2,3,4
}

✓ 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
    - pebble(*i*, .)에 pebble(*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

```
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] ;

        return  max { peb[n, p] } ;
}
```

무시

기껏 n 바퀴

p와 양립하는 패턴 q

p =1,2,3,4

기껏 3 가지

✓복잡도 : Θ(n)

n * 4 * 3 = Θ(n)