알고리즘

9장 동적 프로그래밍
(dynamic programming)

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. 행렬 경로 문제
3. 돌 놓기 문제
4. 행렬 곱셈 순서 문제
5. 최장 공통 부분 순서

학습목표

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

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. 행렬 경로 문제
3. 돌 놓기 문제
4. 행렬 곱셈 순서 문제
5. 최장 공통 부분 순서

배 경

- 재귀적 해법
  - 큰 문제에 닮음꼴의 작은 문제가 깃든다.
  - 잘 쓰면 보약, 잘못 쓰면 맹독
    - 관계중심으로 파악함으로써 문제를 간명하게 볼 수 있다.
    - 재귀적 해법을 사용하면 심한 중복 호출이 일어나는 경우가 있다. 이런 경우에는 재귀적 해법을 사용하면 안된다.

배 경

- 큰 문제의 해답에 작은 문제의 해답이 포함되어 있는 문제
  - 관계 중심으로 파악하여 문제를 간명하게 정의 가능

    예) factorial을 보통의 방식으로 정의하면 n! = 1·2·3···(n−1)·n

    이지만, 관계의 관점에서 정의하면 n! = n(n−1)!
  - 이런 종류의 문제 해결에 재귀 알고리즘(recursive algorithm)을 사용할 수 있는데,
    - 재귀적 구현이 바람직한 경우도 있지만,
    - 심한 중복 호출이 일어나서 재귀적 구현이 매우 비효율적인 경우도 있다.

➔ 재귀적 구현시 심한 중복 호출 문제가 발생하는 경우, 동적 프로그래밍(dynamic programming) 으로 해결

재귀적 해법의 빛과 그림자

- 모든 재귀 알고리즘에서 심한 중복이 발생하는 것은 아니다.
- 재귀적 해법이 바람직한 예 ➔ 재귀적으로 구현해도 심한 중복 호출이 발생하지 않음
  - 퀵정렬, 병합정렬
  - 계승(factorial) 구하기
  - 그래프의 깊이우선탐색(DFS)
  - …
- 재귀적 해법이 치명적인 예 ➔ 심한 중복 호출이 발생하므로 동적 프로그래밍이 필요
  - 피보나치수 구하기
  - 행렬곱셈 최적순서 구하기
  - …

도입문제: 피보나치수 구하기

- 피보나치 수열의 정의

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

  $f(1) = f(2) = 1$

- 아주 간단한 문제지만 Dynamic Programming (DP)의 동기와 구현을 설명해주는 예이다.

피보나치수 구하기 - Recursive Algorithm

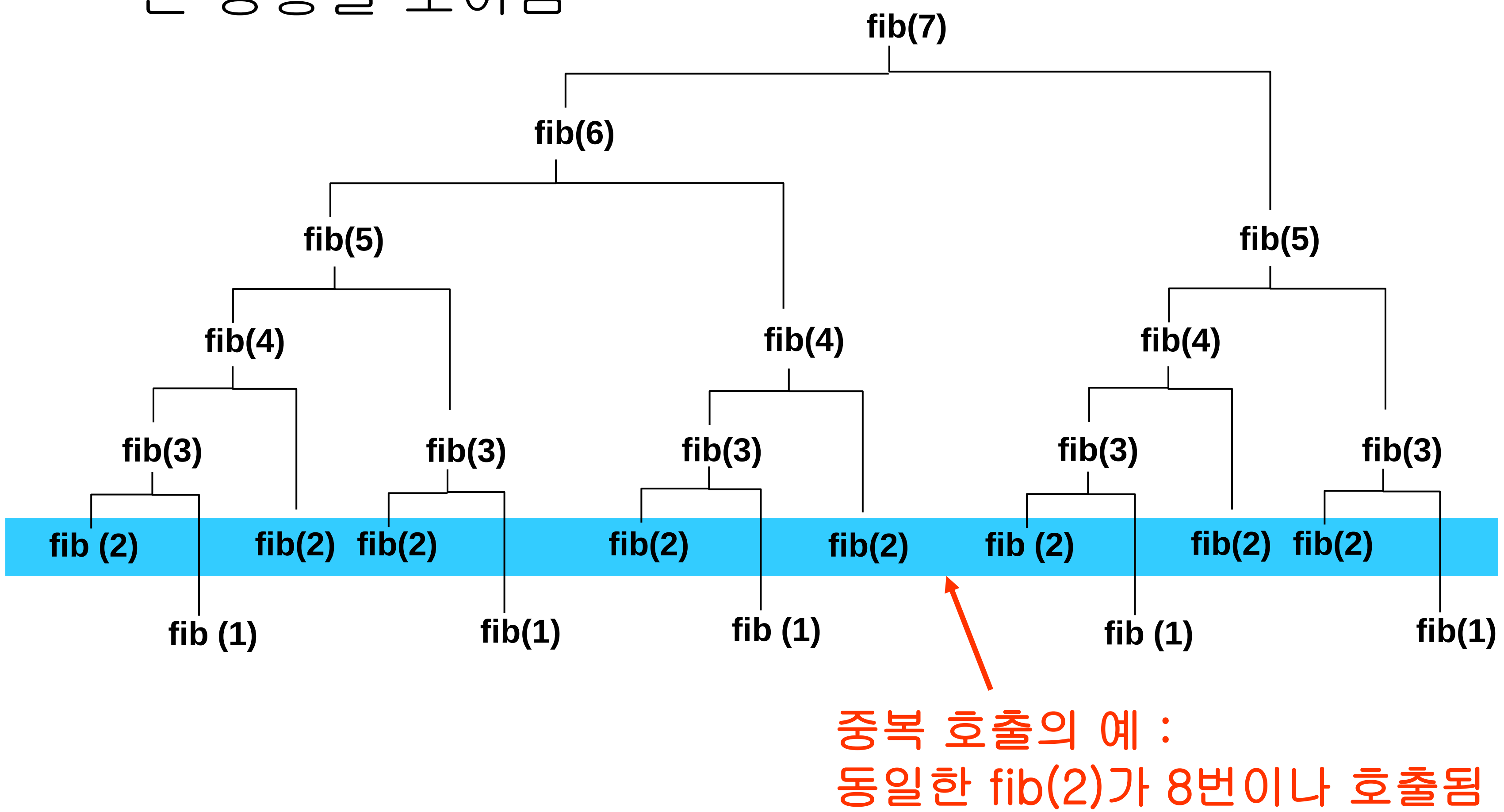
$$f(n) = f(n\text{-}1) + f(n\text{-}2)$$
$$f(1) = f(2) = 1$$

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

✓ 엄청난 중복 호출이 일어난다.

피보나치수 구하기의 Call Tree

- 재귀적 구현으로 동일한 문제가 지나치게 중복 호출되는 상황을 보여줌



피보나치수 구하기 – 재귀적 해법

- 문제점:
  - 피보나치수 계산을 위한 재귀 알고리즘(recursive algorithm)은 지나친 중복 호출이 발생한다.
  - 예를 들어 fib(7) 계산시 fib(2)를 8번이나 호출
- 해결책: 동적 프로그래밍
  - 호출 결과를 처음 얻었을 때 저장해 두고 필요할 때 사용하면 됨
  - 동적 프로그래밍의 핵심은 부분 결과를 저장하면서 해를 구해 나가는 것
  - 두 가지 버전의 피보나치수 구하기 DP 알고리즘을 살펴보자.

두 가지 버전의 동적 프로그래밍 알고리즘

- Top-down 방식
  - 재귀적으로 구현하되, 호출된 적이 있으면 메모해 두어 중복 호출 문제 해결
  - 즉, 함수의 앞부분에 이미 해결한 적이 있는 문제인지를 체크하는 부분을 두고, 이미 한 번 해결한 문제이면 함수를 더 이상 진행하지 않고 테이블에 있는 해를 리턴
- Bottom-up 방식
  - 작은 문제의 해부터 테이블에 저장해가면서 이들을 이용해 큰 문제들의 해를 구해 나간다.
  - 더 흔히 사용되는 방식

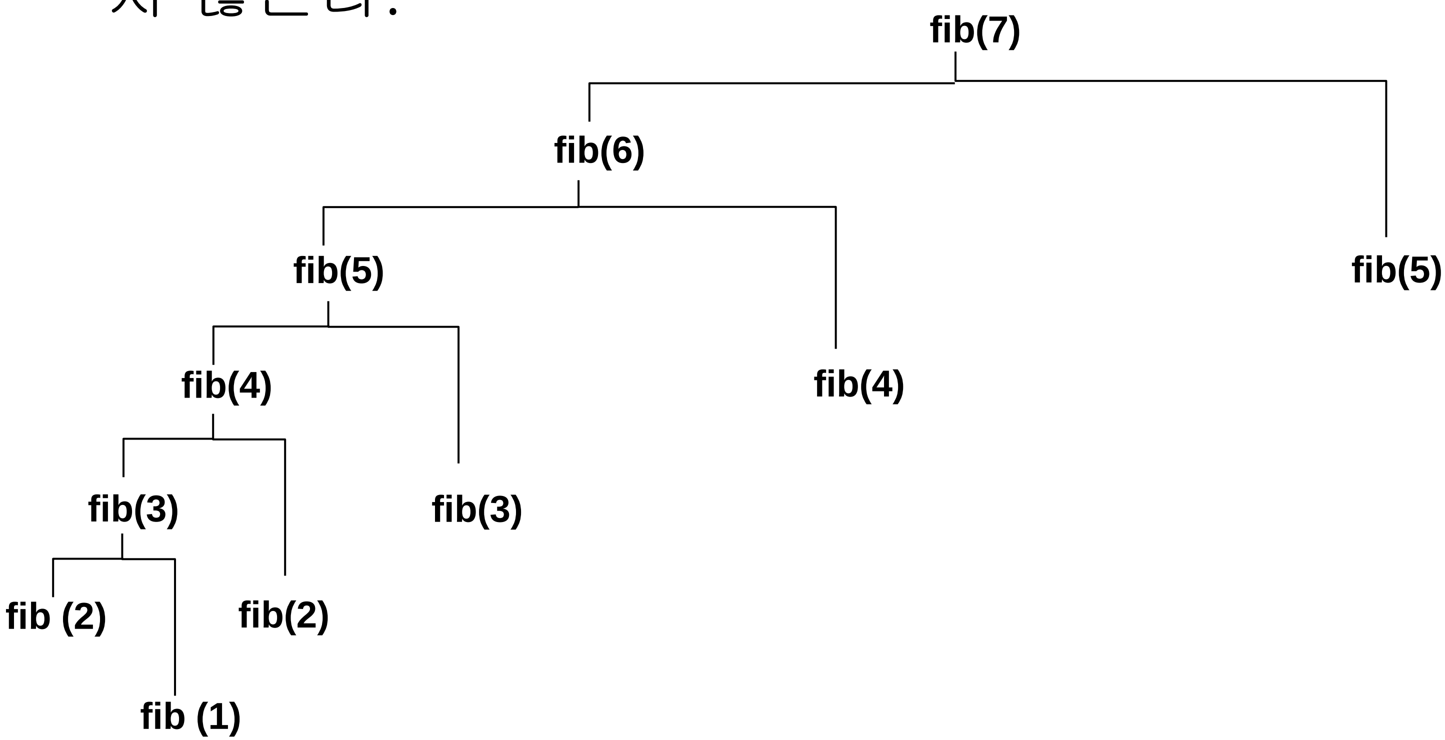
동적 프로그래밍(DP) 알고리즘 1

- Top-down 방식
- 재귀적으로 구현하되, 호출된 적이 있으면 배열 f[ ]에 메모해 두어(memoization = 메모리에 저장) 중복 호출 문제 해결
  - 배열 f[ ]의 원소는 0으로 초기화됨
  - f[i]의 값이 0이면 fib(i)가 아직 한 번도 수행되지 않았음을 의미

```
fib(n)
{
    if (f[n] ≠ 0) then return f[n];   ▷ 호출된 적이 있으면
    else {
       if (n = 1 or n = 2)
          then f[n] ← 1;
          else  f[n] ← fib(n-1) + fib(n-2);
       return f[n];
    }
}
```

DP 알고리즘 1의 Call Tree

- 재귀적 구현이지만 동일한 문제가 심하게 중복 호출되지 않는다.



동적 프로그래밍(DP) 알고리즘 2

- Bottom-up 방식 – 더 흔히 사용됨

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

✓ Time complexity : $\Theta(n)$. 즉, 선형 시간에 끝난다.

동적 프로그래밍의 적용 요건

- 최적 부분구조(optimal substructure)
  - 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함됨
- 재귀 호출시 중복(overlapping recursive calls)
  - 재귀적 해법으로 풀면 같은 문제에 대한 재귀 호출이 심하게 중복됨

➡ 이 두 가지 모두 해당되면 동적 프로그래밍이 해결책이다.

동적 프로그래밍 예

- 행렬 경로 문제
- 돌 놓기 문제
- 행렬 곱셈 순서 문제
- 최장 공통 부분 순서

요약

- 동적 프로그래밍(dynamic programming)은 최적 부분 구조를 가지며 재귀적으로 구현했을 때 중복 호출로 심각한 비효율이 발생하는 문제의 해결에 적합한 기법이다.
- 상향식 동적 프로그래밍은 작은 문제의 해부터 테이블에 저장해가면서 이들을 이용해 큰 문제들의 해를 구해 나간다.
- 메모하기 방식의 동적 프로그래밍(하향식 동적 프로그래밍)은 재귀적으로 구현하되 함수의 앞부분에서 이미 해결한 적이 있는 문제인지 체크하여 이미 해결한 문제이면 함수를 더 이상 진행하지 않고 테이블에 있는 해를 리턴한다.

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. **행렬 경로 문제**
3. 돌 놓기 문제
4. 행렬 곱셈 순서 문제
5. 최장 공통 부분 순서

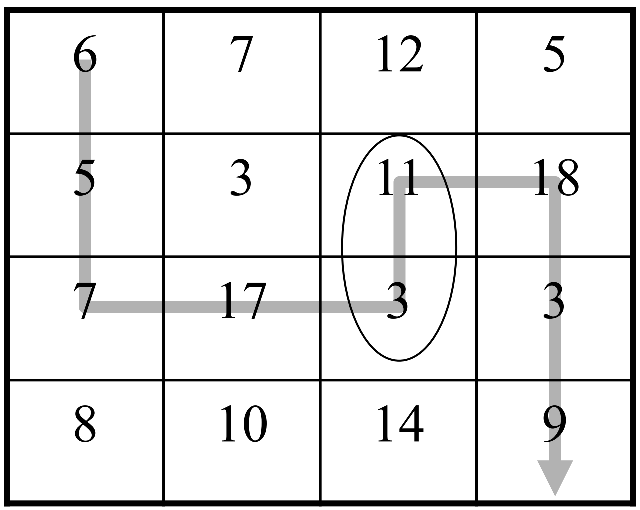
행렬 경로 문제

- 양수 원소들로 구성된 $n \times n$ 행렬이 주어지고, 행렬의 좌상단에서 시작하여 우하단까지 이동한다.

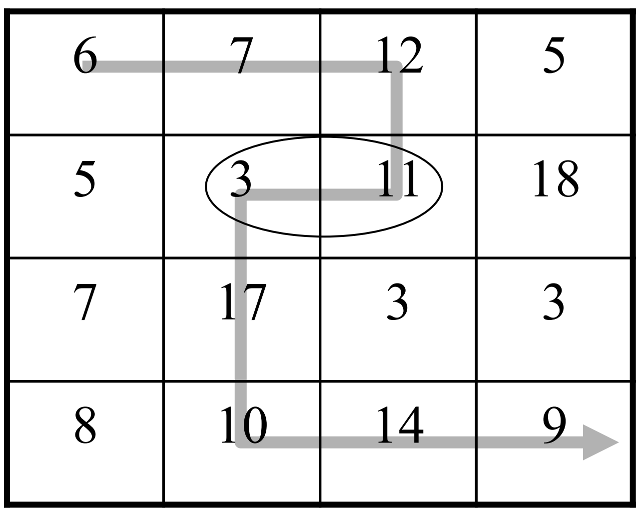
6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

- 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 <u>최대</u>가 되도록 한다.

불법 이동의 예

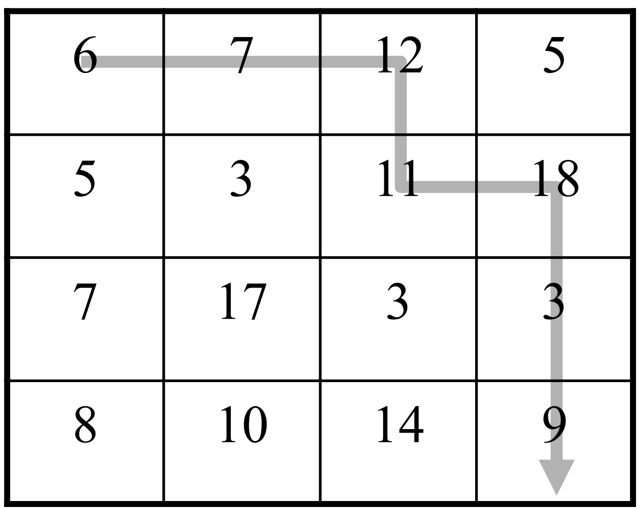


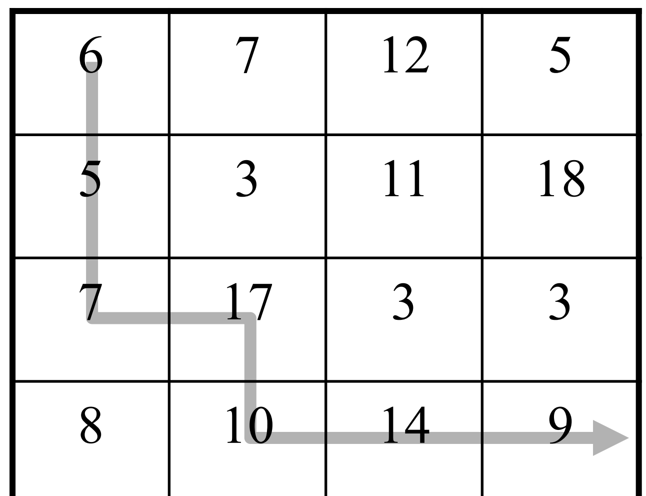
불법 이동 (상향)



불법 이동 (좌향)

유효한 이동의 예





최적 부분구조의 재귀적 관계

c_{ij} 는 (1,1)에서 (i, j)에 이르는 최고점수

m_{ij} 는 행렬의 원소 (i, j) 의 값

$$c_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ m_{ij} + \max\{c_{i,j-1}, c_{i-1,j}\} & \text{otherwise} \end{cases}$$

최종적으로 $c_{n,n}$ 이 답이다.

행렬 경로 문제 – Recursive Algorithm

matrixPath(i, j) ▷ (1,1)에서 (i, j)에 이르는 최고점수

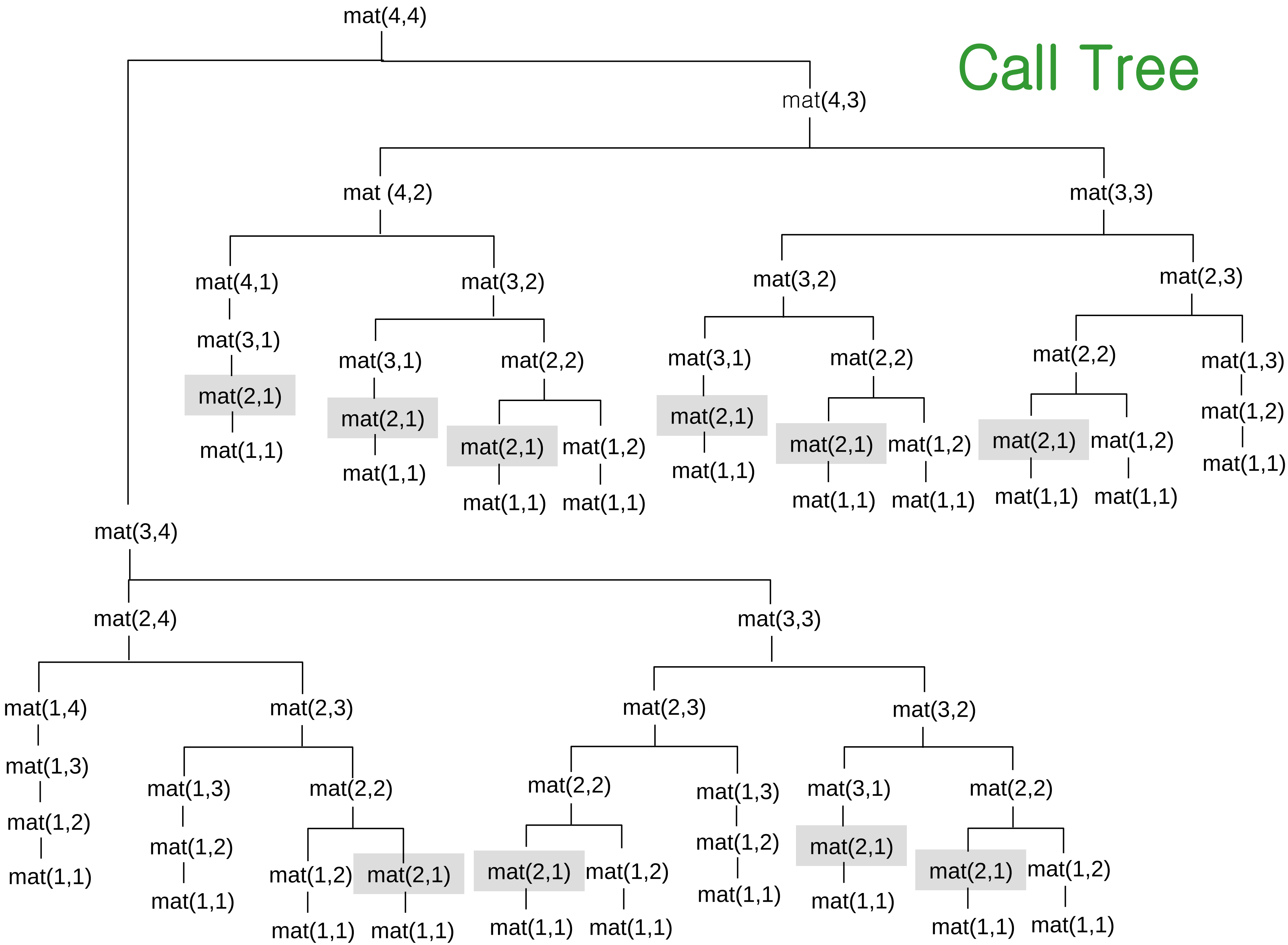
{

if (i = 0 **or** j = 0) **then return** 0;

else return (m_{ij} + max(matrixPath(i-1, j), matrixPath(i, j-1)));

}

Call Tree



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)

재귀 알고리즘의 중복 호출

matrixPath(i, j)	matrixPath(2, 1)의 중복 호출 횟수
matrixPath(2, 2)	1
matrixPath(3, 3)	13
matrixPath(4, 4)	10
matrixPath(5, 5)	35
matrixPath(6, 6)	126
matrixPath(7, 7)	462
matrixPath(8, 8)	1714
matrixPath(9, 9)	6435

행렬 경로 문제 – 동적 프로그래밍 알고리즘

부분 문제의 답을 배열 c에 저장
$c[i, j]$: (1, 1)에서 (i, j)에 이르는 최고점수

matrixPath(n) ▷ (1,1)에서 (n, n)에 이르는 최고점수
{
  for $i \leftarrow 0$ **to** n
    $c[i, 0] \leftarrow 0;$
  for $j \leftarrow 1$ **to** n
    $c[0, j] \leftarrow 0;$
  for $i \leftarrow 1$ **to** n
    for $j \leftarrow 1$ **to** n
      $c[i, j] \leftarrow m_{ij} + \max(c[i\text{-}1, j], c[i, j\text{-}1]);$
  return $c[n, n];$
}

✓Time complexity: $\Theta(n^2)$
✓행렬의 원소 수에 대해서는 선형 시간

간단한 예

3 x 3 행렬 m

i \ j	1	2	3
1	5	15	3
2	1	4	1
3	10	2	5

배열 c

i \ j	0	1	2	3
0				
1				
2				
3				

문제: 다음과 같은 4 x 4 크기의 행렬 경로 문제를 푸는 과정과 경로의 최대 점수를 구하시오.

4 x 4 행렬 m

i \ j	1	2	3	4
1	5	1	2	3
2	2	1	3	2
3	2	2	1	1
4	2	3	2	3

최대 점수 =

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. 행렬 경로 문제
3. **돌 놓기 문제**
4. 행렬 곱셈 순서 문제
5. 최장 공통 부분 순서

돌 놓기 문제

- 3 x n 테이블의 각 칸에 숫자(양수 또는 음수)가 기록되어 있고, 이 테이블의 칸에 돌을 놓는다.

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

- 돌을 놓는 방법 (제약조건)
  - 가로나 세로로 인접한 두 칸에 동시에 돌을 놓을 수 없다.
  - 각 열에는 적어도 하나의 조약돌을 놓아야 한다.
- 목표: 돌이 놓인 자리에 있는 수의 합이 최대가 되도록 돌을 놓는다.

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

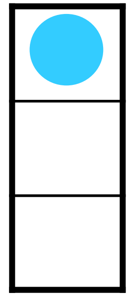
합법적이지 않은 예

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!

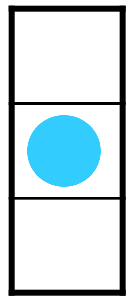
가능한 패턴

패턴 1:



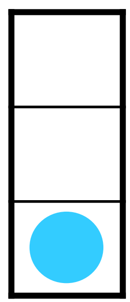
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

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

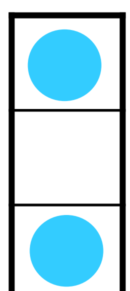
패턴 3:



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

임의의 열을 채울 수 있는
패턴은 네 가지뿐이다.

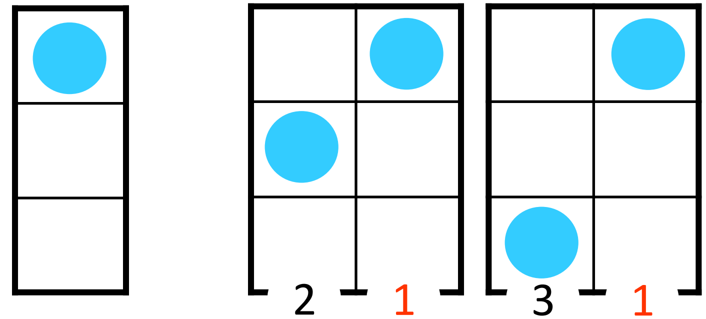
패턴 4:



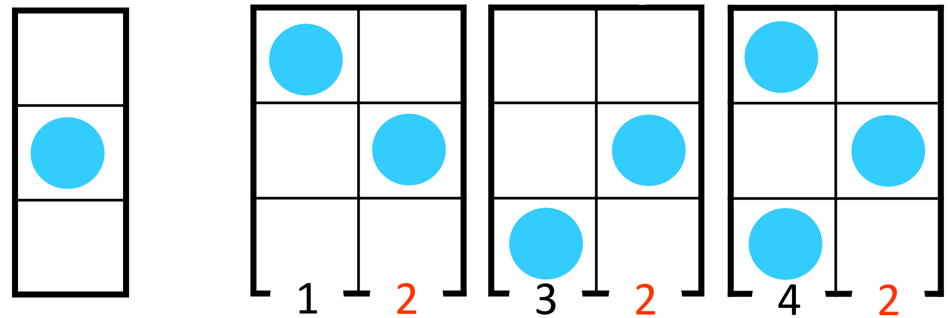
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

서로 양립할 수 있는 패턴들

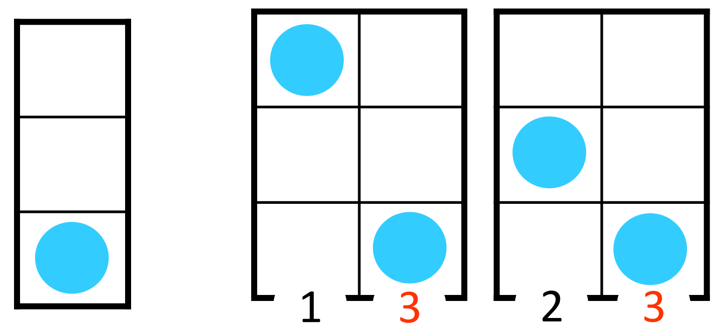
패턴 1:



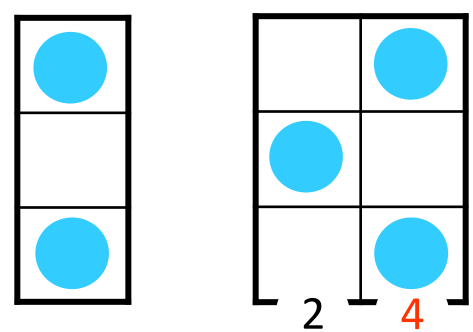
패턴 2:



패턴 3:



패턴 4:



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

i 열과 *i*-1열의 관계를 파악해 보자.

			i-1	*i*			
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

i 열이 패턴 1인 경우, *i*-1열은 ...

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

패턴 2이거나,

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

패턴 3이다.

i 열과 *i*-1열의 관계를 파악해 보자.

			i-1	*i*			
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

i 열이 패턴 2인 경우, *i*-1열은 ...

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

패턴 1이거나,

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

패턴 3이거나,

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

패턴 4이다.

i 열과 *i*-1열의 관계를 파악해 보자.

			i-1	*i*			
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4	8	-2	9	4

i 열이 패턴 3인 경우, *i*-1열은 ...

i 열과 *i*-1열의 관계를 파악해 보자.

i-1 *i*

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

i 열이 패턴 4인 경우, *i*-1열은 ...

최적 부분구조의 재귀적 관계

c_{ip} 는 i열이 패턴 p로 놓일 때의 최고점수
w_{ip} 는 i열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합

$$c_{ip} = \begin{cases} w_{1p} & \text{if } i = 1 \\ \max\limits_{\text{p와 양립하는 패턴 q}}\{c_{i-1,q}\} + w_{ip} & \text{if } i > 1 \end{cases}$$

최종적으로 c_{n1}, c_{n2} , c_{n3}, c_{n4} 중 가장 큰 것이 답이다.

돌 놓기 문제 - Recursive Algorithm

```
pebble(i, p)
▷ i열이 패턴 p로 놓일 때의 i열까지의 최대 점수 합 구하기
▷ w_ip : i열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합. p∈{1, 2, 3, 4}
{
    if (i = 1)
            then return w_1p;
            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_ip) ;
            }
}
```

pebbleSum(n)

▷ n 열까지 돌을 놓은 방법 중 최대 점수 합 구하기

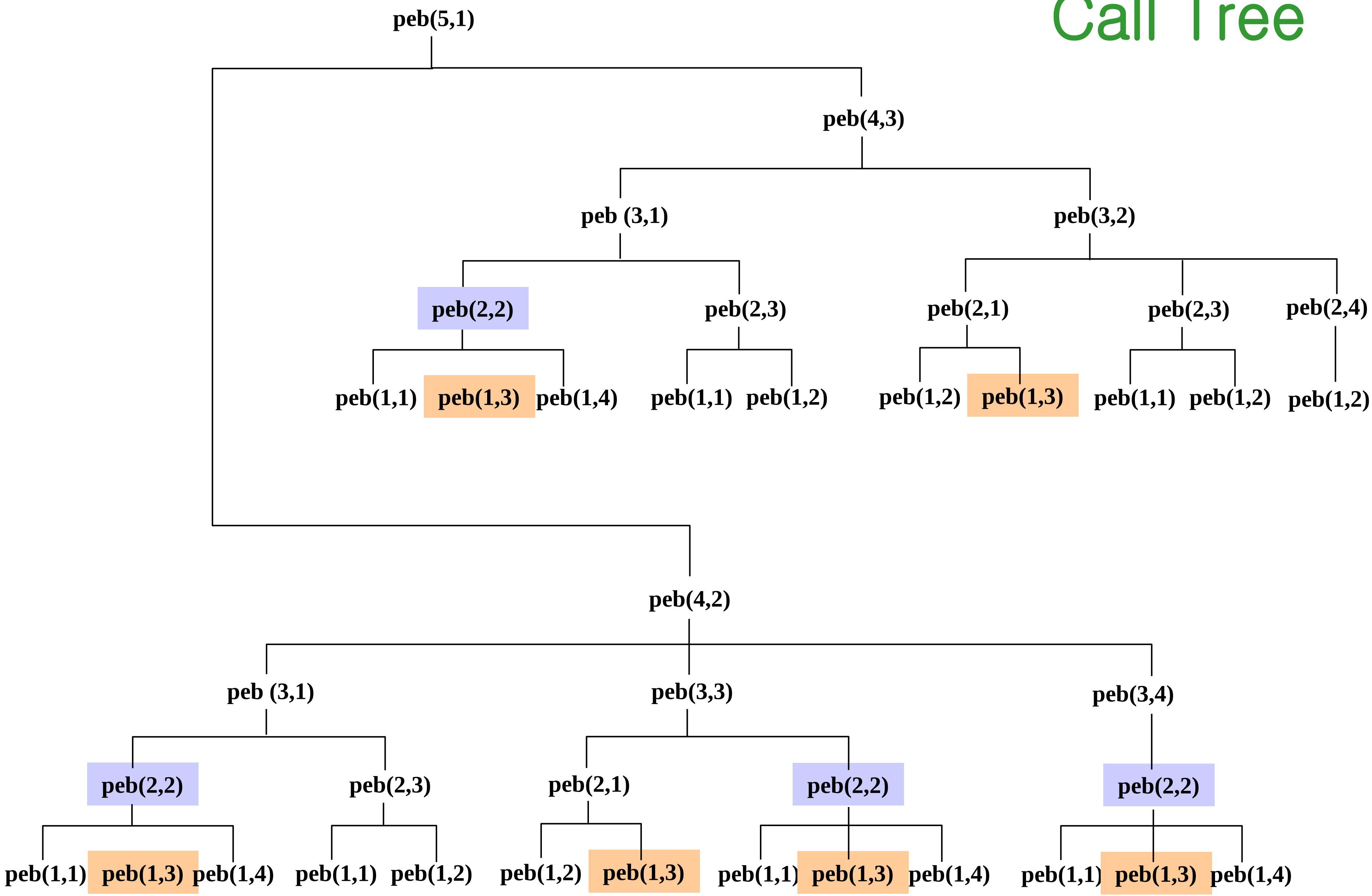
{

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

}

✓ pebble(n, 1), …, pebble(n, 4) 중 최대값이 돌 놓기 문제의 최종 답

Call Tree


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)

돌 놓기 문제 – 동적 프로그래밍 적용

- DP의 요건 만족
  - Optimal substructure
    - pebble(i, .)에 pebble(i-1, .)이 포함됨
    - 즉, 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함됨
  - Overlapping recursive calls
    - 재귀적 알고리즘에 중복 호출 심함

돌 놓기 문제 – 동적 프로그래밍 알고리즘

부분 문제의 답을 4 x n 배열 peb에 저장
peb[i, p] : i열이 패턴 p로 놓일 때, 1열부터 i열까지의 최대 점수합

pebble(n)
{
  for $p \leftarrow 1$ **to** 4
    peb[1, p] $\leftarrow w_{1p}$;
  for $i \leftarrow 2$ **to** n
    for $p \leftarrow 1$ **to** 4
      peb[i, p] $\leftarrow w_{i\,p} + \max\limits_{p\text{와 양립하는 패턴 } q} \{ \text{peb}[i-1, q] \}$;

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

✓Time complexity: $\Theta(n)$

복잡도 분석

pebble(n)
{

for $p \leftarrow 1$ **to** 4

$\quad$ peb[1, p] $\leftarrow w_{1p}$;

for $i \leftarrow 2$ **to** n

$\quad$ **for** $p \leftarrow 1$ **to** 4

$\quad\quad$ peb[i, p] $\leftarrow w_{i\,p} + \max$ { peb[i-1, q] } ;

$\quad\quad$ p와 양립하는 패턴 q

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

}

무시

4번 반복

n 번 이하 반복

3 가지 이하

✓ 시간 복잡도 : $\boldsymbol{\Theta(n)}$

간단한 예

3 x 5 테이블

i	1	2	3	4	5
	5	3	-20	10	10
	2	5	10	5	-5
	10	6	1	-10	5

peb

p \ i	1	2	3	4	5
1					
2					
3					
4					

이해를 돕기 위해 배열 peb[i, p]를 행과 열을 바꾸어 나타냄

문제: 다음과 같은 3 x 8 테이블에 대한 돌 놓기 문제에서 최대 점수를 구하시오.

3 x 8 테이블

i	1	2	3	4	5	6	7	8
	1	6	12	-5	5	3	9	2
	-14	9	14	9	7	13	6	4
	6	11	7	4	8	-2	7	3

최대 점수 =

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. 행렬 경로 문제
3. 돌 놓기 문제
4. **행렬 곱셈 순서 문제**
5. 최장 공통 부분 순서

행렬 곱셈 순서 문제

- 행렬 A, B, C
  - (AB)C = A(BC)
- 예: A:10 x 100, B:100 x 5, C:5 x 50
  - (AB)C: 7500번의 스칼라 곱셈 필요
  - A(BC): 75000번의 스칼라 곱셈 필요
- A_1, A_2, A_3, …, A_n을 곱하는 최적의 순서는?
  - 총 n-1회의 행렬 곱셈을 어떤 순서로 할 것인가?

재귀적 관계

- 마지막으로 행렬 곱셈이 수행되는 상황
  - $n-1$ 가지 가능성
    - $(A_1 \cdots A_{n-1})A_n$
    - $(A_1 \cdots A_{n-2})\ (A_{n-1}A_n)$
    - . . .
    - $(A_1A_2)(A_3 \cdots A_n)$
    - $A_1(A_2 \cdots A_n)$
  - 어느 경우가 가장 곱셈 연산을 적게 하는가?
- n개의 행렬 곱셈에 괄호 묶는 총 가지 수: $\Omega(2^n)$

최적 부분구조의 재귀적 관계

- ✓ C_{ij} : $A_i, A_{i+1}, \ldots, A_j$를 곱하는 최소 비용
- ✓ A_k의 차원: $p_{k\text{-}1} \times p_k$

$$C_{ij} = \begin{cases} 0 & \text{if } i=j \\ \min\limits_{i \le k \le j-1} \{ \underbrace{C_{i,k} + C_{k+1,\,j} + p_{i-1}p_k p_j}_{\text{두 행렬 곱셈 } (A_i \cdots A_k)(A_{k+1} \cdots A_j) \text{ 의 최소비용}} \} & \text{if } i<j \end{cases}$$

최종적으로 C_{1n} 이 답이다.

행렬 곱셈 순서 문제 – Recursive Algorithm

rMatrixChain(i, j)
▷ 행렬곱 $A_i \cdot A_{i+1} \cdots A_j$를 구하는 최소 비용을 구한다.
{
    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!
✓ Time complexity : $\Omega(2^n)$

행렬 곱셈 순서 문제 – DP 알고리즘

부분 문제의 답을 n x n 배열 m에 저장
m[i, j] : A_i, A_{i+1}, …, A_j를 곱하는 최소 비용

```
matrixChain(n)
{
    for i ← 1 to n
          m[i, i] ← 0;        ▷ 행렬이 하나뿐인 경우의 비용은 0
    for r ← 1 to n-1          ▷ r+1 은 문제의 크기
          for i ← 1 to n-r {
                j ← i+r;
                m[i, j] ← min_{i ≤ k ≤ j-1} {m[i, k] + m[k+1, j] + p_{i-1}p_k p_j};
          }
    return m[1, n];
}
```

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

✓ Time complexity: $\Theta(n^3)$

간단한 예

A_1 A_2 A_3 A_4 를 계산하는 최소 스칼라 곱셈 수는?
3x4, 4x5, 5x2, 2x10

$<p_0, p_1, p_2, p_3, p_4>$
$= <3, 4, 5, 2, 10>$

m_{14}를 구하면 됨

m

i \ j	1	2	3	4
1				
2				
3				
4				

문제: 행렬 곱셈 $A_1A_2A_3A_4$ 를 계산할 때 최소 스칼라 곱셈 개수를 구하시오.

단, A_1, A_2, A_3, A_4 의 크기는 각각 5x2, 2x2, 2x10, 10x3 이다.

과정과 답을 적을 것

학습내용

1. 어떤 문제를 동적 프로그래밍으로 푸는가
2. 행렬 경로 문제
3. 돌 놓기 문제
4. 행렬 곱셈 순서 문제
5. **최장 공통 부분 순서**

최장 공통 부분순서(LCS)

- 두 문자열에 공통적으로 들어있는 공통 부분순서 중 가장 긴 것을 찾는다.
- Subsequence의 예
  - <bcdb>는 문자열 <abcbdab>의 subsequence
- Common subsequence의 예
  - <bca>는 문자열 <abcbdab>와 <bdcaba>의 common subsequence
- Longest common subsequence(LCS)
  - Common subsequence들 중 가장 긴 것
  - 예: <bcba>는 string <abcbdab>와 <bdcaba>의 LCS

최장 공통 부분순서(LCS)

- 문자열 abcd의 부분순서(subsequence)
- 문자열 abcd와 aabbdd의 공통 부분순서(common subsequence)
- 문자열 abcd와 aabbdd 의 최장 공통 부분순서(LCS: longest common subsequence)

최적 부분구조의 재귀적 관계

- 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-1}과 Y_{n-1}의 LCS의 길이보다 1이 크다.
  - $x_m \neq y_n$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.

✓ c_{ij} : 두 문자열 $X_i = <x_1x_2 \ldots x_i>$과 $Y_j = <y_1y_2 \ldots y_j>$의 LCS 길이

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

최종적으로 c_{mn} 이 답이다.

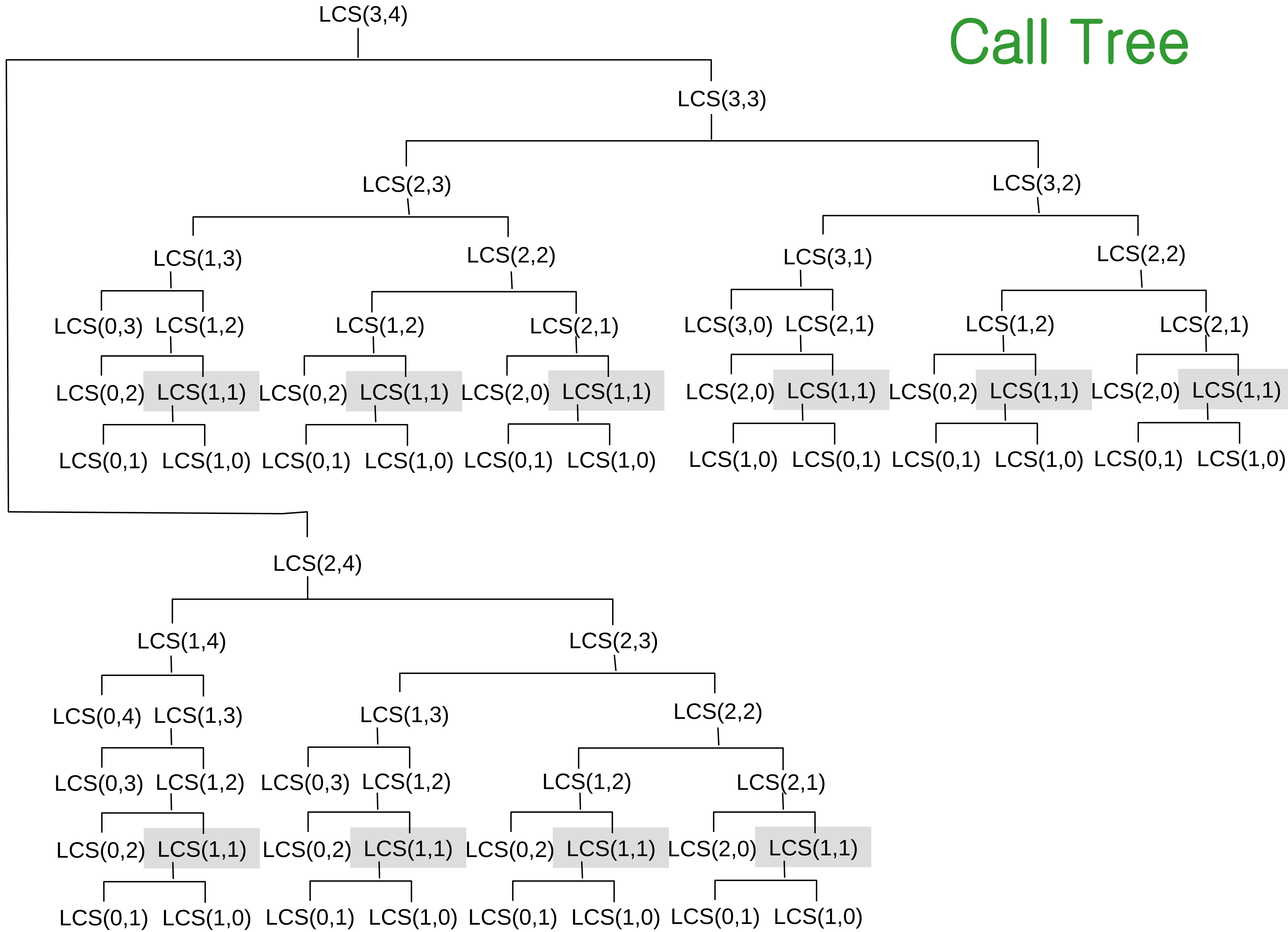
최장 공통 부분순서 - Recursive Algorithm

LCS(m, n) ▷ 두 문자열 X_m과 Y_n의 LCS 길이를 구한다.

```
{
    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!

Call Tree


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)

최장 공통 부분순서 - DP 알고리즘

LCS(m, n) ▷ 두 문자열 X_m과 Y_n의 LCS 길이를 구한다.

```
{
    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 (x_i = y_j) 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];
}
```

부분 문제의 답을 배열 C에 저장
C[i, j] : X_i과 Y_j의 LCS 길이를 저장

✓ Time complexity: $\Theta(mn)$

간단한 예

X_4 = baba

Y_3 = aab

C

j	0	1	2	3
i 0				
1				
2				
3				
4				

문제: 다음 두 문자열 X_4 = abcd 와 Y_4 = acbc 의 LCS 길이를 구하시오. 과정과 답을 적을 것

요약

- 동적 프로그래밍은 최적 부분 구조를 가지며 재귀적으로 구현했을 때 중복 호출로 심각한 비효율이 발생하는 문제의 해결에 적합한 기법
- 동적 프로그래밍의 예
  - 행렬 경로 문제
  - 돌 놓기 문제
  - 행렬 곱셈 순서 문제
  - 최장 공통 부분순서