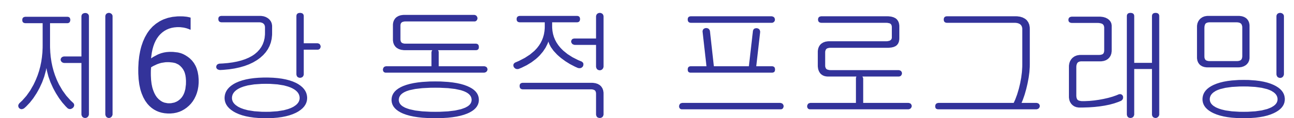


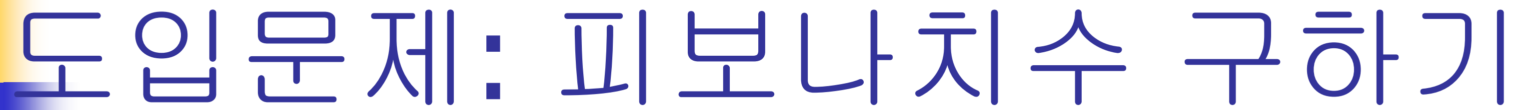
김 승 주
정보보호연구소
성균관대학교 정보통신공학부
http://www.security.re.kr/

배경

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

재귀적 해법의 빛과 그림자

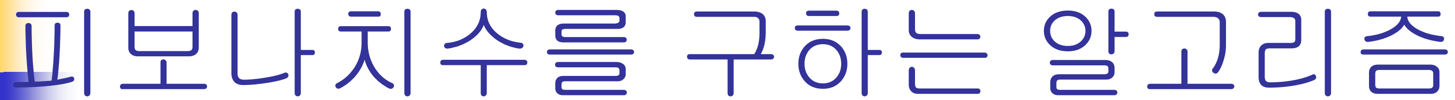
- 재귀적 해법이 바람직한 예
  - 퀵정렬, 병합정렬 등의 정렬 알고리즘
  - 계승(factorial) 구하기
  - 그래프의 DFS
  - …
- 재귀적 해법이 치명적인 예
  - 피보나치수 구하기
  - 행렬곱셈 최적순서 구하기
  - …



도입문제: 피보나치수 구하기

- $f(n) = f(n-1) + f(n-2)$
- $f(1) = f(2) = 1$

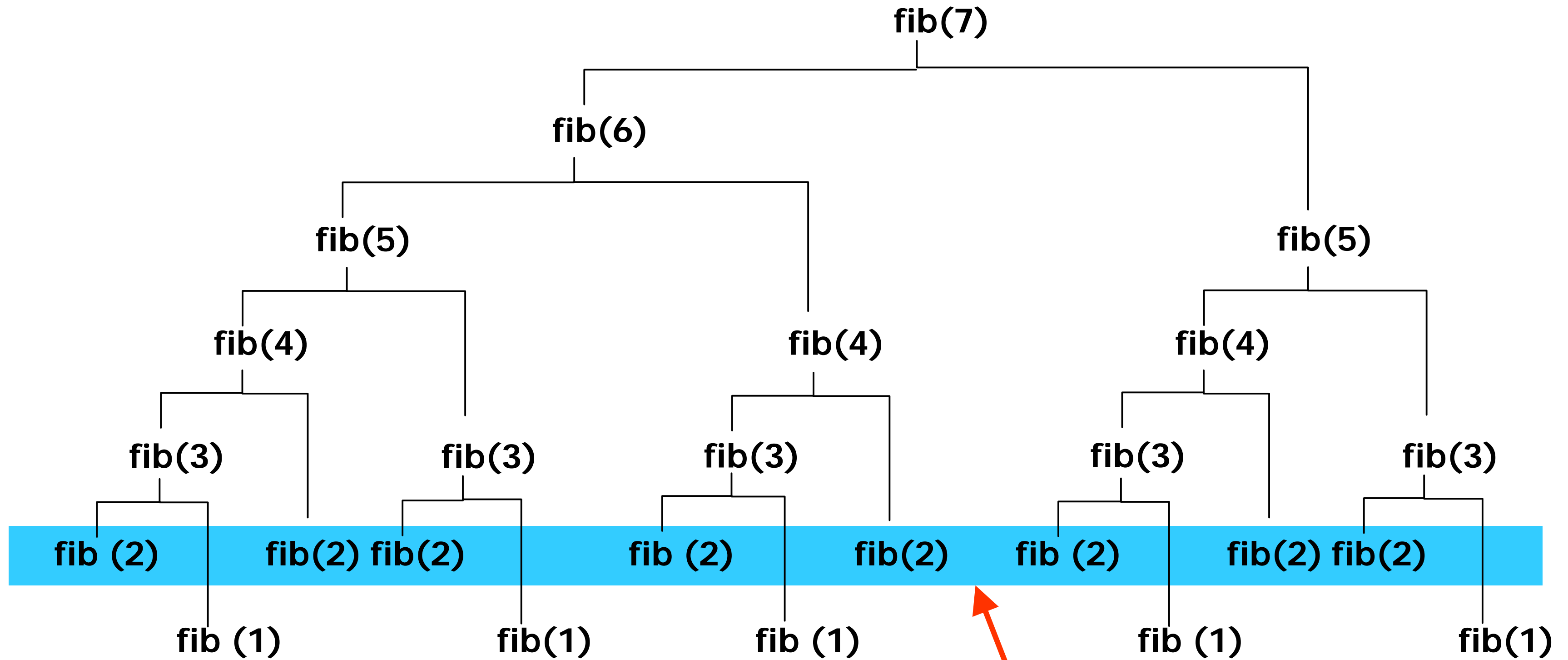
- 아주 간단한 문제지만
  - 동적 프로그래밍의 동기와 구현이 다 포함되어 있다




```
1.  fib(n)
2.  {
3.          if (n = 1 or n = 2)
4.              then return 1;
5.              else return (fib(n-1) + fib(n-2));
6.  }
```

- 위의 피보나치수를 구하는 재귀 알고리즘에는 엄청난 중복 호출이 존재한다

피보나치 수열의 호출 트리



피보나치수를 구하는 DP알고리즘

```
1. fibonacci(n)
2. {
3.         f[1] ← f[2] ← 1;
4.         for i ← 3 to n
5.                 f[i] ← f[i-1] + f[i-2];
6.         return f[n];
7. }
```

- 선형시간에 끝난다

동적 프로그래밍의 적용 요건

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

➢ 동적 프로그래밍(Dynamic Programming, DP)이 그 해결책!

[참고] DP와 분할정복

- 동적 프로그래밍과 분할 정복 모두 주어진 문제를 여러 개의 소문제로 분할하여 각 소문제의 해결안을 바탕으로 주어진 문제를 해결하는 기법이다.
- 그러나 분할 정복에서는 분할되는 소문제가 서로 독립적이지만, 동적 프로그래밍에서는 소문제가 독립적이지 않다. 다시 말해서 분할된 소문제간에 중복되는 부분이 있다.
- 그러므로 동적 프로그래밍에서는 소문제의 해를 표 형식으로 저장.

문제예 1: 행렬 경로 문제

- 양 또는 음의 정수 원소들로 구성된 n×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 최대가 되도록 한다

불법 이동의 예

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

불법 이동 (상향)

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

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

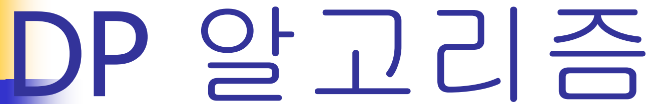
1. matrixPath(i, j) ▷ (i, j)에 이르는 최고점수
2. {
3.   if (i = 1 and j = 1) then return m_{11};
4.   else if (i = 1) then
            return (m_{1j} + matrixPath(1, j-1));
5.   else if (j = 1) then
            return (m_{i1} + matrixPath(i-1, 1));
6.   else return (m_{ij} + (max(matrixPath(i-1, j),
    matrixPath(i, j-1))));
7. }

호출 트리의 예

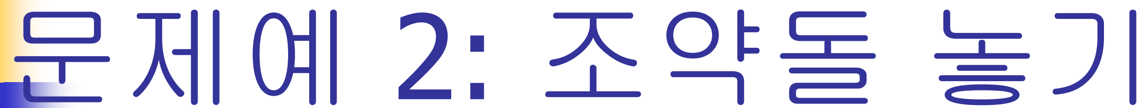
mat(4,4)
- mat(3,4)
  - mat(2,4)
    - mat(1,4)
      - mat(1,3)
        - mat(1,2)
          - mat(1,1)
    - mat(2,3)
      - mat(1,3)
        - mat(1,2)
          - mat(1,1)
      - mat(2,2)
        - mat(1,2)
          - mat(1,1)
        - mat(2,1)
          - mat(1,1)
  - mat(3,3)
    - mat(2,3)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

DP 적용

- DP의 요건 만족
  - 최적 부분구조
    - 원소 (i,j)까지 도달하는 최고점수 = max{(i-1,j)까지 도달하는 최고점수, (i,j-1)까지 도달하는 최고점수} + 원소 (i,j)의 점수
    - 즉, 큰 문제의 최적 솔루션에 작은 문제의 최적 솔루션이 포함됨
  - 재귀호출시 중복
    - 재귀적 알고리즘에 중복 호출 심함



1. matrixPath(n) ▷ (n, n)에 이르는 최고점수
2. {
3.   c[1, 1] ← m_{11};
4.   for i ← 2 to n
5.   c[i, 1] ← m_{i1} + c[i-1, 1];
6.   for j ← 2 to n
7.   c[1, j] ← m_{1j} + c[1, j-1];
8.   for i ← 2 to n
9.   for j ← 2 to n
10.   c[i, j] ← m_{ij} + max(c[i-1, j], c[i, j-1]);
11.   return c[n, n];
12. }



문제예 2: 조약돌 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 예와 불법적인 예

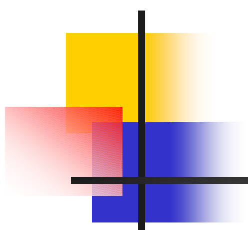
합법적인 예

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:

●

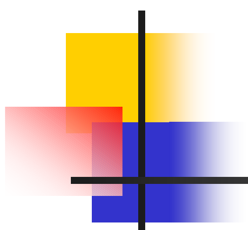
6	7	12	-5	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4 ●	8	-2	9	4

패턴 4:

●
●

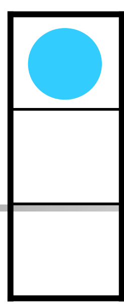
6	7	12	-5 ●	5	3	11	3
-8	10	14	9	7	13	8	5
11	12	7	4 ●	8	-2	9	4

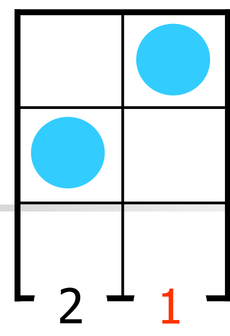
✓ 임의의 열을 채울 수 있는 패턴은 **4**가지뿐이다

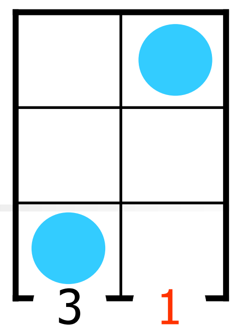


서로 양립할 수 있는 패턴들

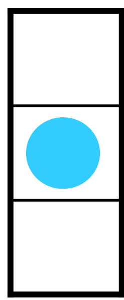
패턴 1:

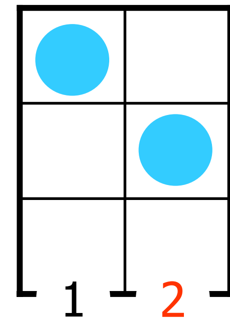


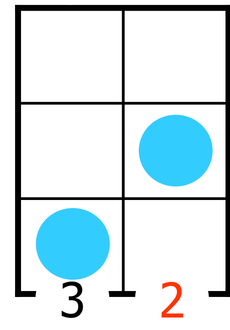


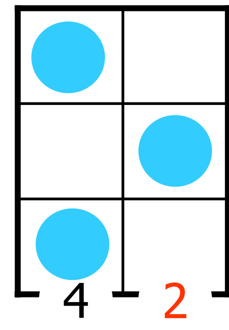


패턴 2:

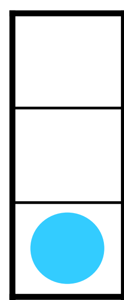


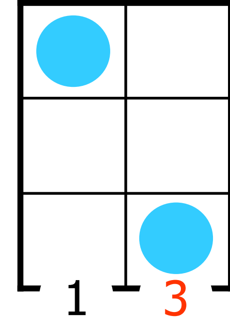


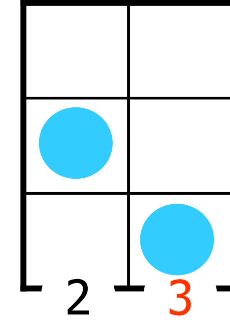




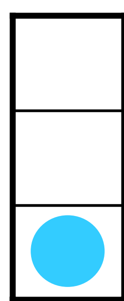
패턴 3:

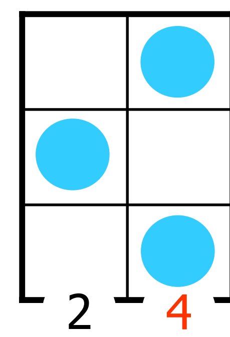






패턴 4:





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

i열과 i-1열의 관계 (1/2)

			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

i열과 i-1열의 관계 (1/2)

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

i열과 i-1열의 관계 (1/2)

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

i열과 i-1열의 관계 (1/2)

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

i열과 i-1열의 관계 (2/2)

- i열이 패턴 1로 놓여있을 경우의 최고점
  - i-1열이 패턴 2로 놓여 있을 경우의 최고점과 i-1열이 패턴 3으로 놓여 있을 경우의 최고점중 큰 것
  - i열에서 패턴 1로 돌이 놓인 곳이 있는 수
- ……
- i열이 패턴 4로 놓여있을 경우의 최고점
  - i-1열이 패턴 2로 놓여 있을 경우의 최고점
  - i열에서 패턴 4로 돌이 놓인 곳이 있는 수

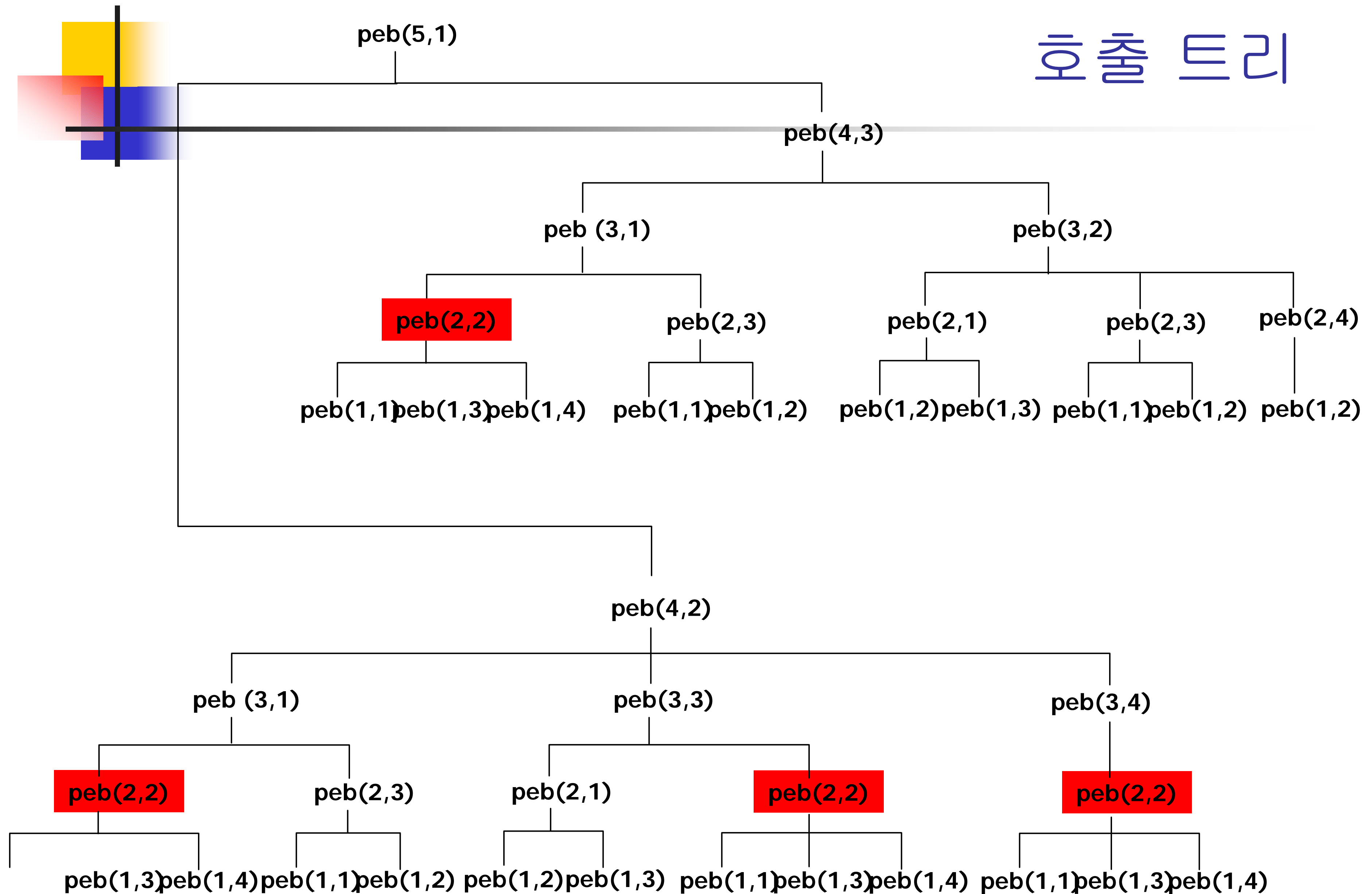
재귀 알고리즘 (1/2)

```
1.  pebble(i, p)
    ▷ i열이 패턴 p로 놓일 때의 i열까지의 최대 점수 합 구하기
    ▷ w[i, p] : i 열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합. p ∈ {1, 2, 3, 4}
2.  {
3.      if (i = 1)
4.          then return w[1, p] ;
5.          else {
6.                  max ← −∞;
7.                  for q ← 1 to 4 {
8.                          if (패턴 q가 패턴 p와 양립)
9.                              then {
10.                                     tmp ← pebble(i−1, q);
11.                                     if (tmp > max) then max ← tmp;
12.                             }
13.                 }
14.                 return (max + w[i, p]);
15.         }
16. }
```

```
1. pebbleSum(n)
   ▷ n 열까지 조약돌을 놓은 방법 중 최대 점수 합 구하기
2. {
3.       return  max {pebble(n, p)};
4. }
```

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

호출 트리



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

```
1. pebble(n)
2. {
3.      for p ← 1 to 4
4.            peb[1, p] ← w[1, p];
5.      for i ← 2 to n
6.            for p ← 1 to 4
7.                  peb[i, p] ← max{peb[i-1, q]} + w[i, p];
                                p와 양립하는 패턴 q
8.      return max{peb[n, p]};
                 p =1,2,3,4
9. }
```

- 복잡도 : O(n)

복잡도 분석

```
1. pebble(n)
2. {
3.     for p ← 1 to 4
4.         peb[1, p] ← w[1, p];
5.     for i ← 2 to n
6.         for p ← 1 to 4
7.             peb[i, p] ← max{peb[i-1, q]} + w[i, p];
8.     return  max{peb[n, p]};
               p =1,2,3,4
9. }
```

무시

기껏 n 바퀴

기껏 4 바퀴

p와 양립하는 패턴 q

기껏 3 가지

- 복잡도 : O(n)

n * 4 * 3 = O(n)

문제 예 3: 행렬 곱셈 순서

- 행렬 A, B, C
  - (AB)C = A(BC)
- 예: A:10 × 100, B:100 × 5, C:5 × 50
  - (AB)C: 7,500번의 곱셈 필요
  - A(BC): 75,000번의 곱셈 필요
- $A_1, A_2, A_3, \ldots, A_n$을 곱하는 최적의 순서는?

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?

변수 정의

- c_{ij}: 행렬 $A_i, \ldots, A_j$의 곱 $A_i \ldots A_j$를 계산하는 최소 비용
- A_k의 차원: $p_{k-1} \cdot p_k$

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

c_{ik}: $(A_i \ldots A_k)$를 위한 최소비용

$c_{k+1,j}$: $(A_{k+1} \ldots A_j)$를 위한 최소비용

$p_{i-1}p_kp_j$: $(A_i \ldots A_k)$와 $(A_{k+1} \ldots A_j)$를 곱하는 비용

재귀적 구현

```
1.  rMatrixChain(i, j)   ▷ 행렬곱 Ai...Aj를 구하는 최소 비용 구하기
2.  {
3.      if (i = j) then return 0;   ▷ 행렬이 하나뿐인 경우의 비용은 0
4.      min ← ∞;
5.      for k ← i to j-1 {
6.          q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + p_{i-1}p_kp_j;
7.          if (q < min) then min ← q;
8.      }
9.      return min;
10. }
```

- 엄청난 중복 호출이 발생한다!

동적 프로그래밍

```
1.  matrixChain(i, j)
2.  {
3.      for i ← 1 to n
4.          m[i, i] ← 0; ▷ 행렬이 하나뿐인 경우의 비용은 0
5.      for r ← 1 to n-1    ▷ r: 문제 크기를 결정하는 변수, 문제의 크기 = r+1
6.          for i ← 1 to n-r {
7.              j ← i+r;
8.              m[i, j] ← min{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j};
9.          }
10.     return m[1, n];
    }
```

- 복잡도: $\Sigma_{r=1}^{n-1}\Sigma_{i=1}^{n-r} r = \Theta(n^3)$

문제 예 4: 최장 공통 부분순서

- 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 부분순서(LCS, Longest Common Subsequence)
  - 공통 부분순서들 중 가장 긴 것
  - 예: <bcba>는 문자열 <abcbdab>와 <bdcaba>의 최장 공통 부분순서다

최적 부분구조 (1/2)

- 두 문자열 X_m = <x_1x_2 … x_m>과 Y_n = <y_1y_2 … 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.
    - 즉, (m, n) 크기의 문제의 해가 (m-1, n-1) 크기의 문제의 해를 포함하고 있다.
  - $x_m \neq y_n$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.
    - 즉, (m, n) 크기의 문제의 해가 (m, n-1) 크기의 문제와 (m-1, n) 크기의 문제의 해를 포함하고 있다.

최적 부분구조 (2/2)

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

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

1. LCS(m, n)
   ▷ 두 문자열 X_m과 Y_n의 LCS 길이 구하기
2. {
3.   if (m=0 or n=0) then return 0;
4.   else if (x_m=y_n) then return LCS(m-1,n-1)+1;
5.   else return max(LCS(m-1,n),LCS(m,n-1));
6. }

- 엄청난 중복 호출이 발생한다!

LCS(3,4)

LCS(3,3)

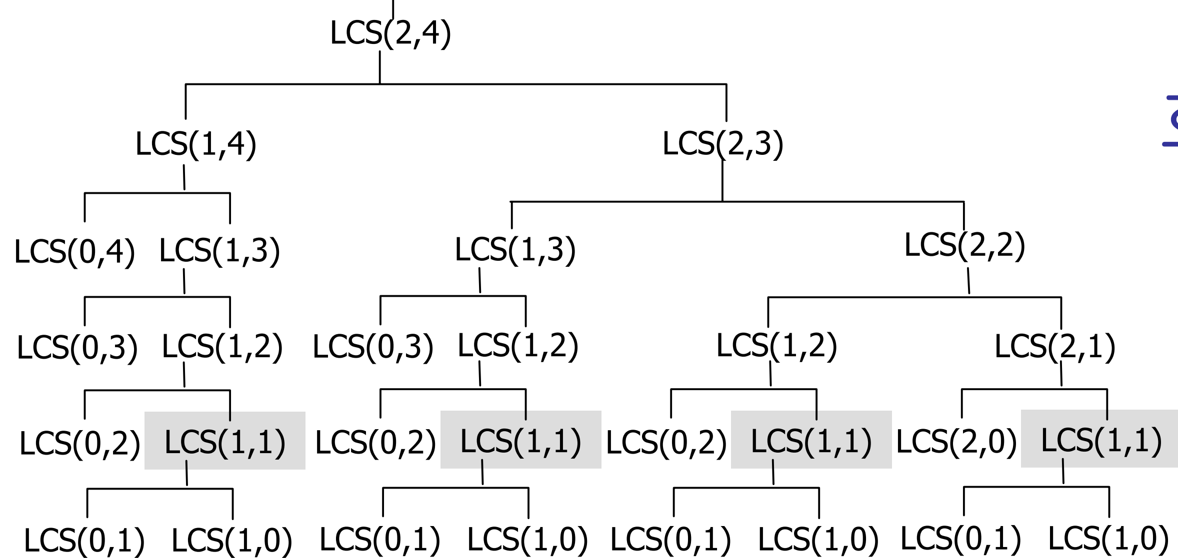
LCS(2,3) LCS(3,2)

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)



호출 트리의 예

동적 프로그래밍

```
1.  LCS(m, n)   ▷ 두 문자열 Xm과 Yn의 LCS 길이 구하기
2.  {
3.      for i ← 0 to m
4.          C[i, 0] ← 0;
5.      for j ← 0 to n
6.          C[0, j] ← 0;
7.      for i ← 1 to m
8.          for j ← 1 to n
9.              if (xi = yj) then C[i, j] ← C[i-1, j-1] + 1;
10.             else C[i, j] ← max(C[i-1, j], C[i, j-1]);
11.     return C[m, n];
12. }
```

- 복잡도: $\Theta(mn)$