Ch.09 동적 프로그래밍

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
  - 관계 중심으로 파악함으로써 문제를 간명하게 볼 수 있다
  - 재귀적 해법을 사용하면 심한 중복 호출이 일어나는 경우가 있다

재귀적 해법의 빛과 그림자

재귀적 해법이 바람직한 예

- 퀵 정렬, 병합 정렬 등의 정렬 알고리즘
- 계승(factorial) 구하기
- 그래프의 DFS
- …

재귀적 해법이 치명적인 예

- 피보나치수 구하기
- 행렬 곱셈 최적 순서 구하기
- …

어떤 문제를 동적 프로그래밍으로 푸는가

피보나치수 구하기

- 아주 간단한 문제지만 동적 프로그래밍의 동기와 구현이 다 포함되어 있다

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

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

- 재귀

알고리즘 9-1 피보나치 수(재귀)

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

✓ 낭비적인 중복 호출이 어마어마하다

어떤 문제를 동적 프로그래밍으로 푸는가

피보나치수 구하기(재귀)

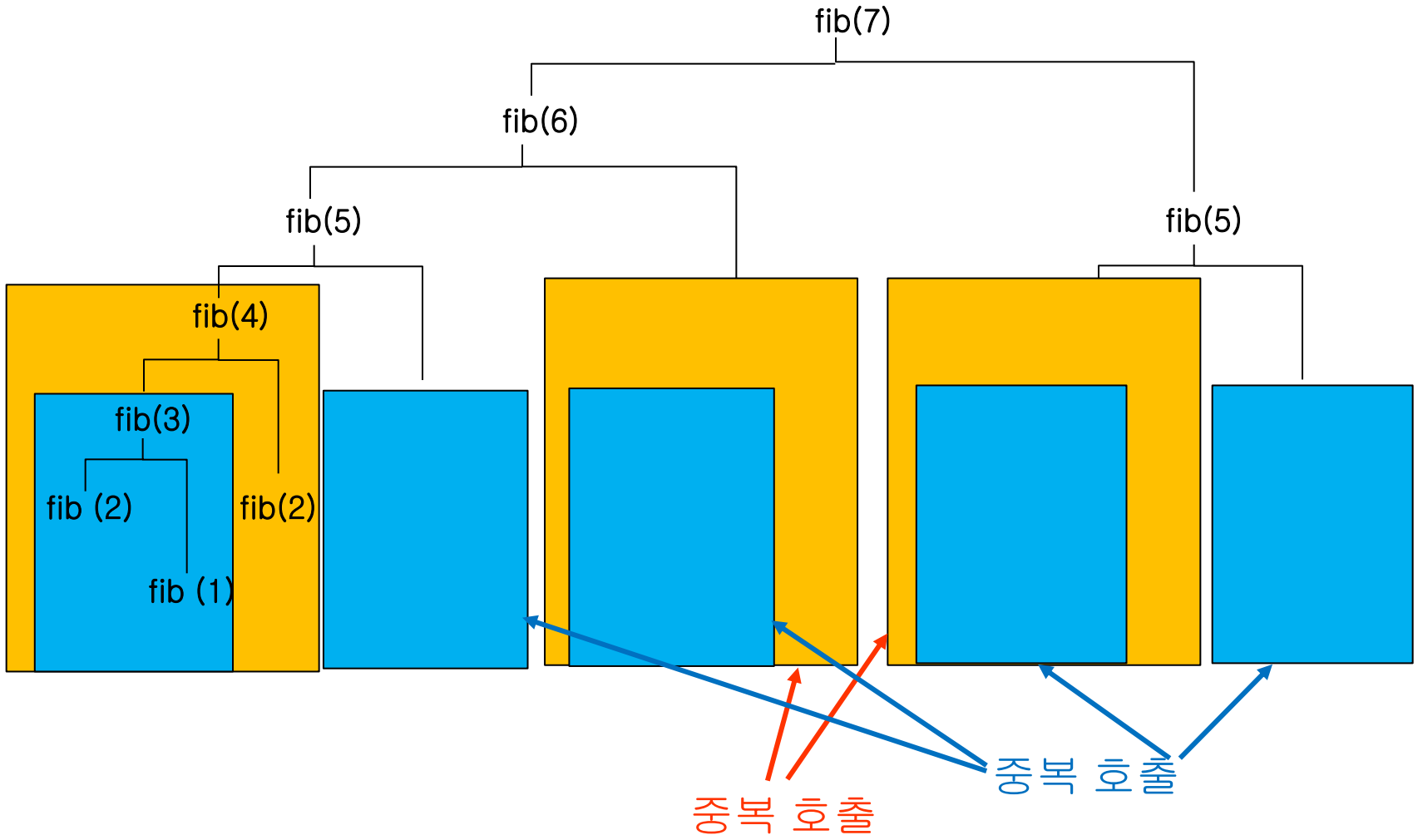


그림 9-1 재귀적 구현으로 동일한 문제가 중복 호출되는 상황을 보여주는 트리 예

표 9-1 fib()에서 문제 크기가 커짐에 따라 중복 호출이 증가하는 모습

수행되는 fib( )	fib(2)의 중복 호출 횟수
fib(3)	1
fib(4)	2
fib(5)	3
fib(6)	5
fib(7)	8
fib(8)	13
fib(9)	21
fib(10)	34

어떤 문제를 동적 프로그래밍으로 푸는가

- **재귀적 fib(100)은 얼마나 걸릴까?**

내 데스크 탑 PC: Pentium 3GHz

fib(50) – 36초

fib(66) – 하루 정도

fib(100) – 3만5천년 정도

fib(136) – 1조년 초과

지수함수적 중복 호출로 인해 이런 치명적인 비효율이 발생한다

어떤 문제를 동적 프로그래밍으로 푸는가

그림 9-2 재귀와 동적 프로그래밍

어떤 문제를 동적 프로그래밍으로 푸는가

피보나치수 구하기(DP)

알고리즘 9-2 피보나치 수(동적 프로그래밍 1)

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

알고리즘 9-3 피보나치 수(동적 프로그래밍 2)

```
▷ 배열 f[]의 모든 원소는 0으로 초기화되어 있다.
▷ f[i]값이 0이면 fib(i)가 아직 한 번도 수행되지 않았음을 의미한다.
fib(n):
❶ if (f[n] ≠ 0) return f[n]
  else
      if (n=1 or n=2) f[n] ← 1
      else f[n] ← fib(n-1)+fib(n-2)
      return f[n]
```

어떤 문제를 동적 프로그래밍으로 푸는가

동적 프로그래밍의 적용 요건

- 최적 부분 구조를 이룬다.
- 재귀적으로 구현했을 때 중복 호출로 심각한 비효율이 발생한다.

행렬 경로 문제

행렬 경로 문제

- 양수 원소들로 구성된 n×n 행렬이 주어지고, 행렬의 좌상단에서 시작하여 우하단까지 이동한다.

이동 규칙

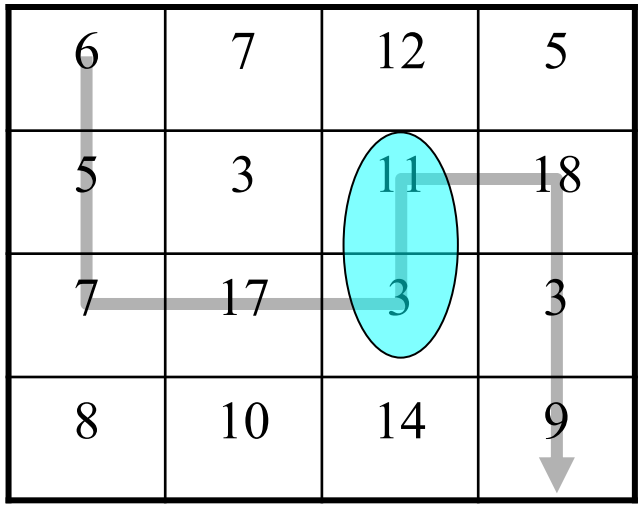
- 오른쪽이나 아래쪽으로만 이동할 수 있다.
- 왼쪽, 위쪽, 대각선 이동은 허용하지 않는다.

목표

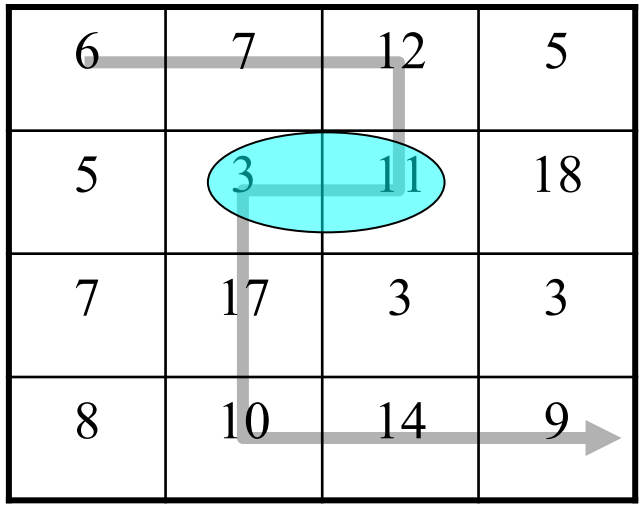
- 행렬의 좌상단에서 시작하여 우하단까지 이동하되, 방문한 칸에 있는 수들을 더한 값이 최소화되도록 한다.

행렬 경로 문제

불법 이동의 예



(a) 불법 이동(상향)

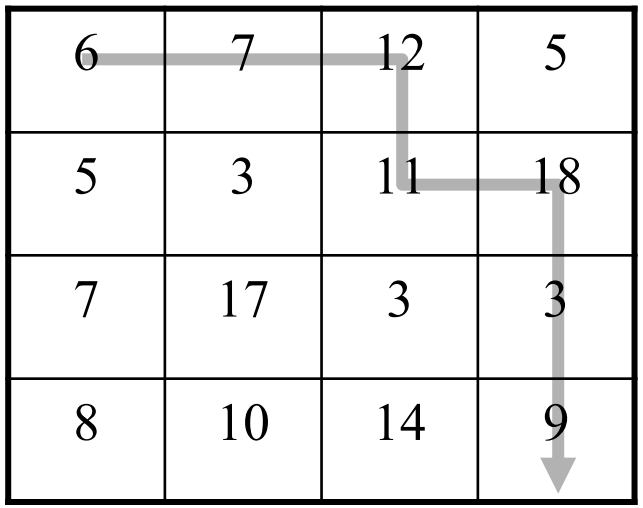


(b) 불법 이동(좌향)

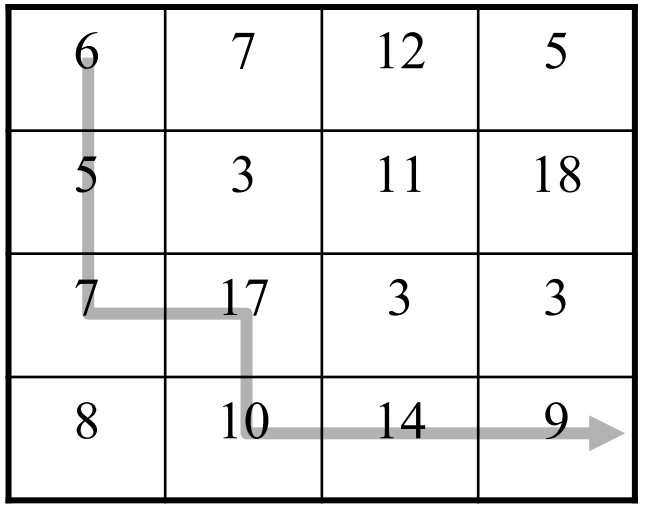
그림 9-3 허용되지 않는 이동과 허용되는 이동의 예

행렬 경로 문제

유효 이동의 예



(c) 유효한 이동

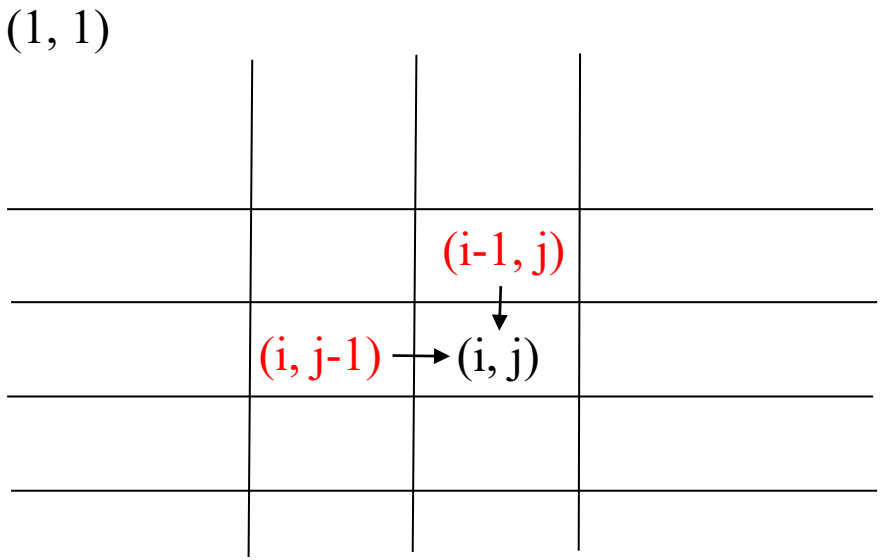


(d) 유효한 이동

그림 9-3 허용되지 않는 이동과 허용되는 이동의 예

행렬 경로 문제

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

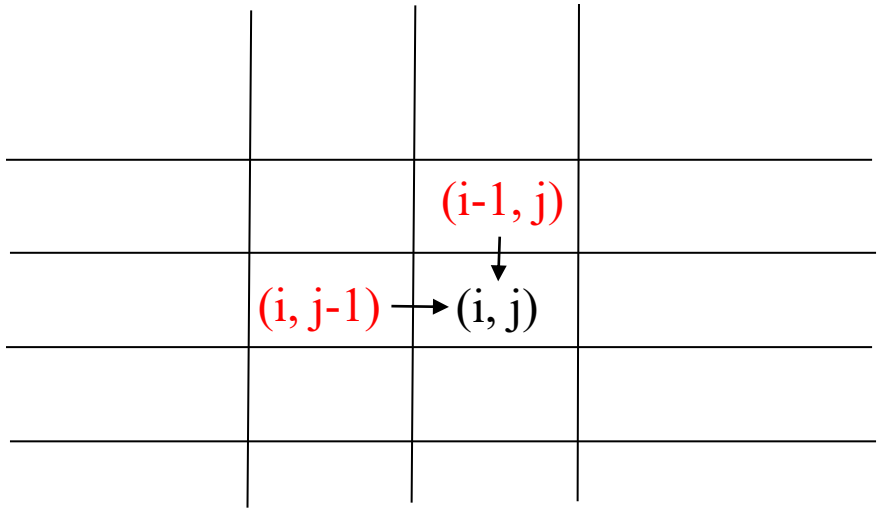


✓ There are just two immediately previous slot to (i, j)

행렬 경로 문제(재귀)

알고리즘 9-4 행렬 경로 문제(재귀)

matrixPath (i, j):
▷ (i, j)에 이르는 최고 점수
    if $(i=0$ or $j=0)$ return 0
    else return $(m_{ij} + (\max(\text{matrixPath}(i-1, j), \text{matrixPath}(i, j-1))))$



행렬 경로 문제(재귀)

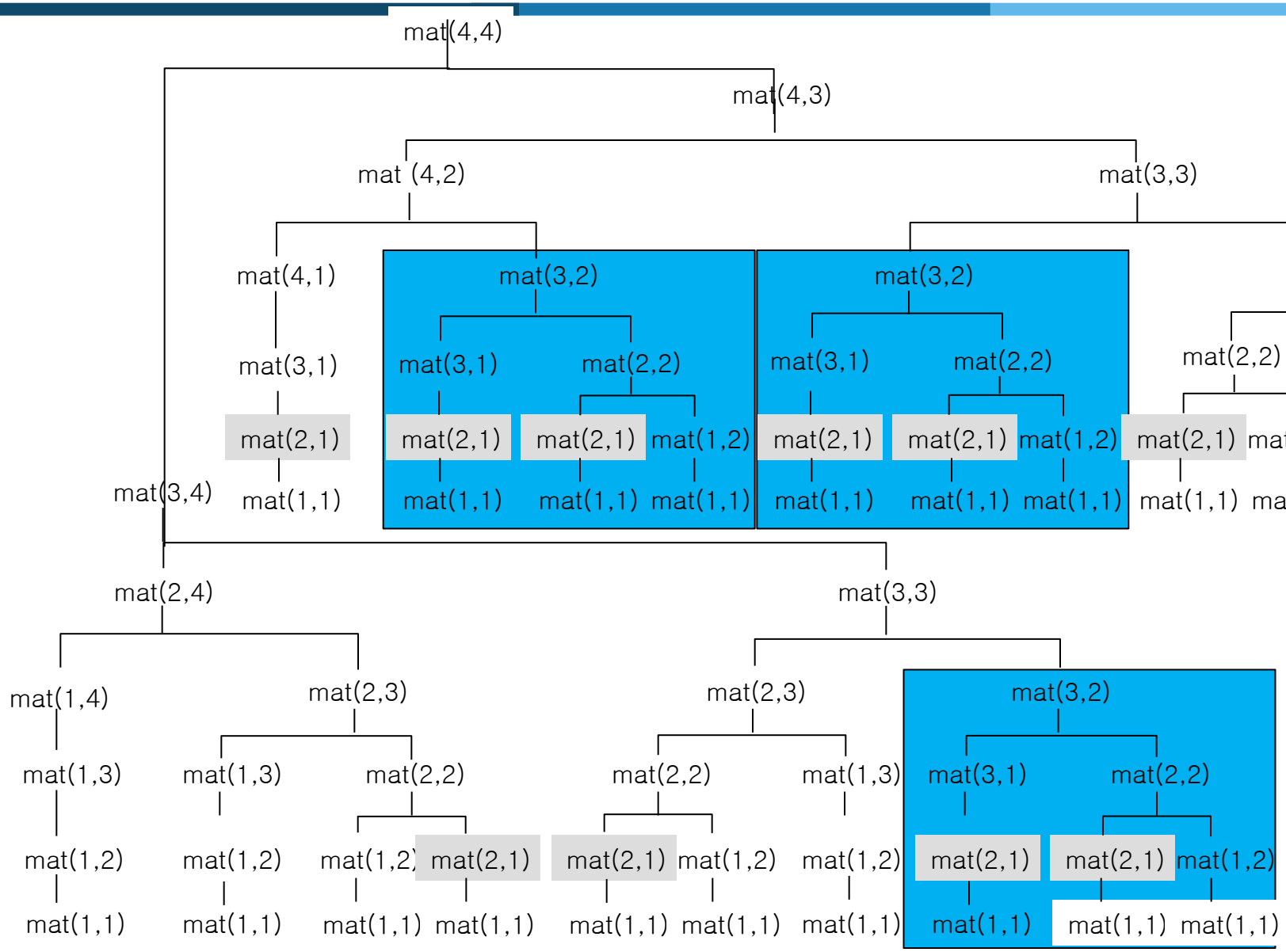


그림 9-8 pebble(5, 1)을 수행하는 과정의 재귀적 호출 관계를 나타내는 트리

표 9-2 matrixPath()에서 문제 크기가 커짐에 따라 중복 호출이 증가하는 모습

수행되는 matrixPath( )	matrixPath(2, 1)의 중복 호출 횟수
matrixPath(2, 2)	1
matrixPath(3, 3)	3
matrixPath(4, 4)	10
matrixPath(5, 5)	35
matrixPath(6, 6)	126
matrixPath(7, 7)	462
matrixPath(8, 8)	1,716
matrixPath(9, 9)	6,435

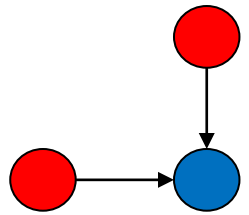
행렬 경로 문제(DP)

알고리즘 9-5 행렬 경로 문제(동적 프로그래밍)

```
matrixPath(n):
▷ (n, n)에 이르는 최고 점수
    for i ← 0 to n
        c[i, 0] ← 0
    for j ← 1 to n
        c[0, j] ← 0
    for i ← 1 to n                                  ❶
        for j ← 1 to n
            ❷ c[i, j] ← m_ij + max(c[i-1, j], c[i, j-1])
    return c[n, n]
```

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

수행 시간: $\Theta(n^2)$



돌 놓기 문제

돌 놓기 문제

- 3×N 테이블의 각 칸에 양 또는 음의 정수가 기록되어 있다.

제한 조건

- 가로나 세로로 인접한 두 칸에 동시에 돌이 놓일 수 없다.
- 각 열에는 적어도 하나 이상의 돌을 놓는다.

목표

- 돌이 놓인 자리에 있는 수의 합을 최대가 되도록 조약돌을 놓는다.

돌 놓기 문제

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

(a) 문제의 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

(b) 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

(c) 합법적이지 않은 예

그림 9–5 돌 놓기 문제의 예와 돌을 놓은 예

돌 놓기 문제

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

그림 9–6 임의의 열에 놓을 수 있는 네 가지 패턴 예

돌 놓기 문제

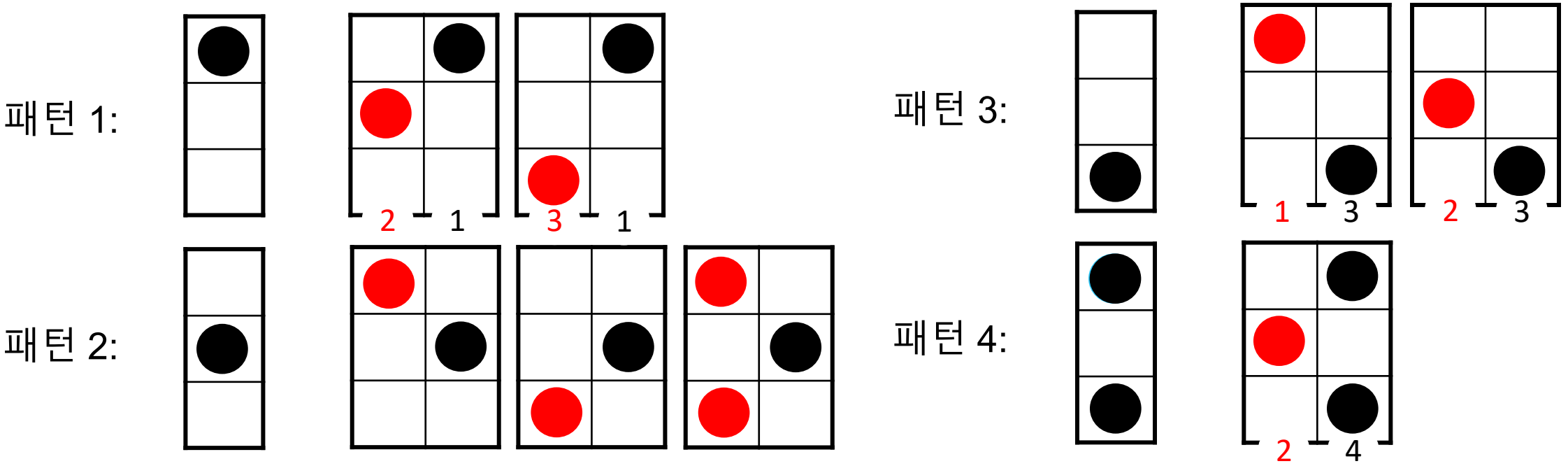


그림 9–7 서로 양립할 수 있는 패턴들

돌 놓기 문제(재귀)

알고리즘 9-6 돌 놓기 문제(재귀)

```
pebble(i, p):
▷ i열이 패턴 p로 놓일 때 최고 점수
▷ w_ip: i열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합, p∈{1, 2, 3, 4}
    if (i=1) return w[1, p]
    else
        max ← −∞
        for q ← 1 to 4
            if (패턴 q가 패턴 p와 양립)
                tmp ← pebble(i−1, q)
                if (tmp > max) max ← tmp
        return (max + w_ip)
```

돌 놓기 문제

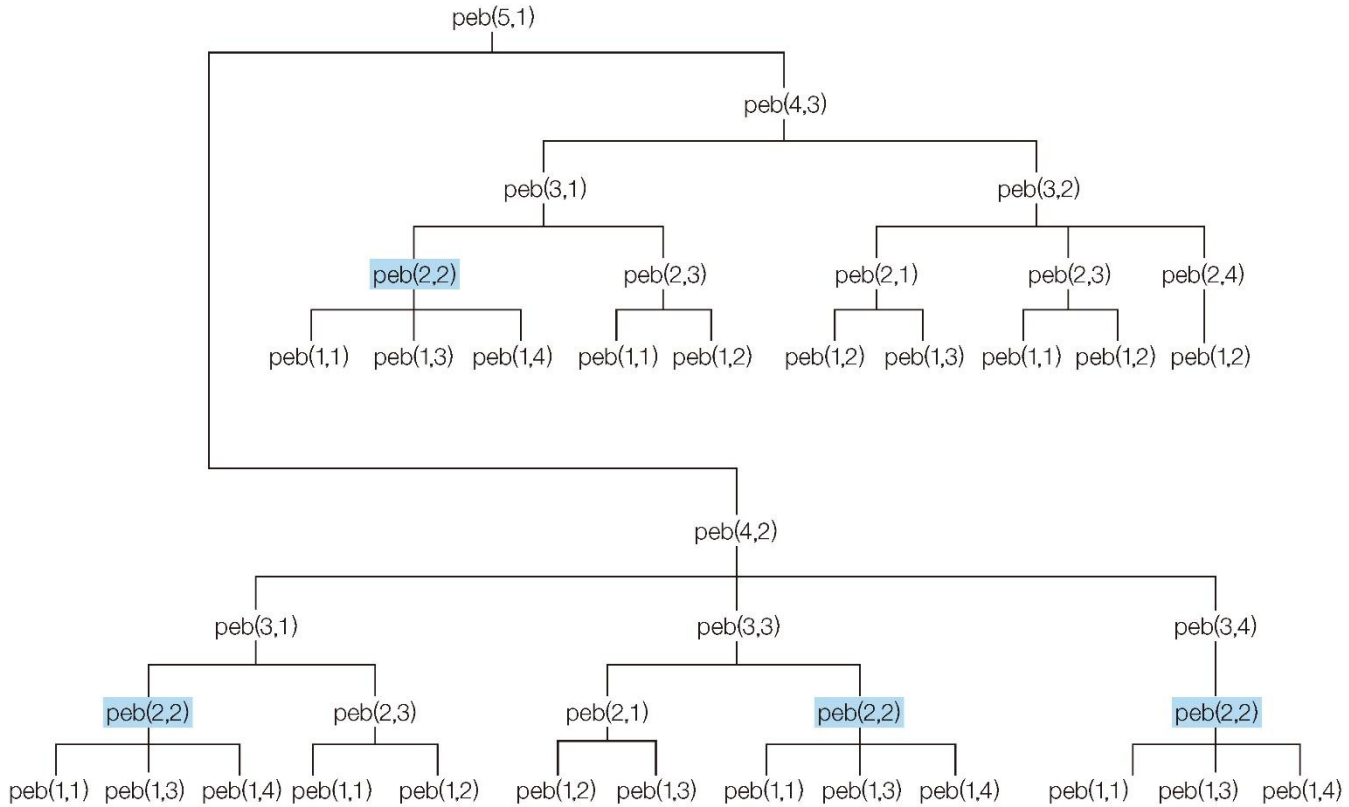


그림 9-8 pebble(5, 1)을 수행하는 과정의 재귀적 호출 관계를 나타내는 트리

표 9-3 pebble()에서 문제가 커짐에 따라 중복 호출의 비율이 증가하는 모습

문제의 크기(n)	부분 문제의 총수	함수 pebble( )의 총 호출 횟수
1	4	4
2	8	12
3	12	30
4	16	68
5	20	152
6	24	332
7	28	726

돌 놓기 문제(DP)

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

돌 놓기 문제(DP)

알고리즘 9-7 돌 놓기 문제(동적 프로그래밍)

pebble(n):

▷ n열까지 돌을 놓을 때 최고 점수

for $p \leftarrow 1$ to 4

  $peb[1, p] \leftarrow w[1, p]$

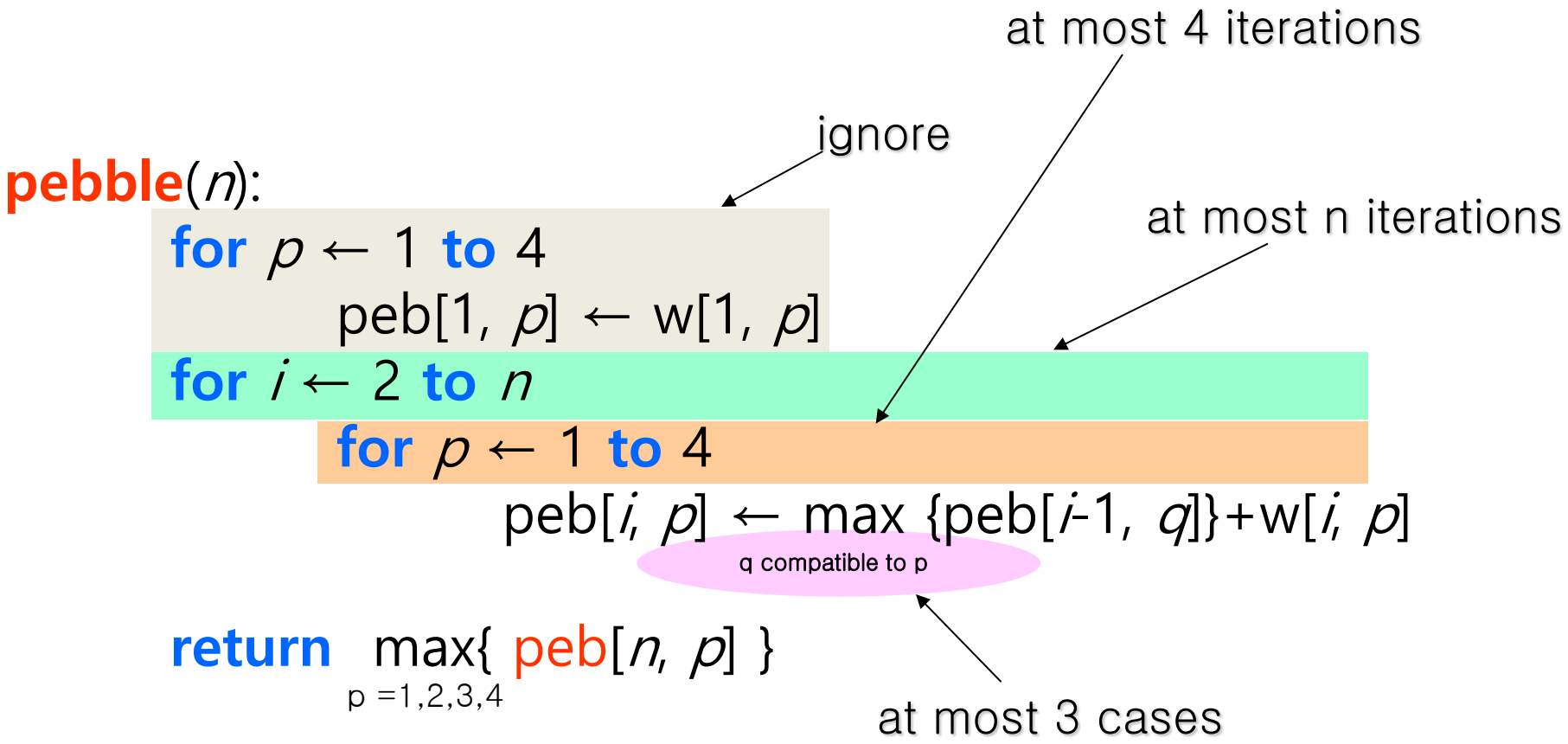
for $i \leftarrow 2$ to n

  for $p \leftarrow 1$ to 4

    $peb[i, p] \leftarrow \max_{p\text{와 양립하는 패턴 } q}\{peb[i-1, q]\} + w_{ip}$

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

돌 놓기 문제(DP)



✓Complexity: $\Theta(n)$

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

돌 놓기 문제(DP)

```
pebble(n)
        for p ← 1 to 4
                peb[1, p] ← w[1, p]
        for i ← 2 to n
                for p ← 1 to 4
①               peb[i, p] ← max {peb[i-1, q]}+w[i, p]
                            q compatible to p

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

✓ Line ①이 수행되는 총 횟수가 시간을 결정한다

✓Complexity: $\Theta(n)$

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: 10 x 100 x 5 + 10 x 5 x 50 = 7,500번의 곱셈 필요
  - A(BC): 100 x 5 x 50 + 10 x 100 x 50 = 75,000번의 곱셈 필요

- **$A_1A_2A_3...A_n$을 곱하는 최적의 순서는?**
  - 총 n-1회의 행렬 곱셈을 어떤 순서로 할 것인가?

행렬 곱셈 순서 문제(재귀)

- **마지막 행렬 곱셈이 수행되는 상황**
  - n-1가지 가능성

$$
\begin{aligned}
&A_1(A_2\cdots A_n)\\
&(A_1A_2)(A_3\cdots A_n)\\
&(A_1A_2A_3)(A_4\cdots A_n)\\
&\cdots\\
&(A_1\cdots A_{n-2})(A_{n-1}A_n)\\
&(A_1\cdots A_{n-1})A_n
\end{aligned}
$$

  - 어느 경우가 가장 매력적인가?

행렬 곱셈 순서 문제(재귀)

- C_{ij}: the minimal cost to compute $A_i \ldots A_j$

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

General form: $(A_i \ldots A_k)\,(A_{k+1} \ldots A_j)$

$A_i(A_{i+1} \quad \ldots \quad A_j)$

…

$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$

…

$(A_i \quad \ldots \quad A_{j-1})A_j$

행렬 곱셈 순서 문제(재귀)

알고리즘 9-8 행렬 곱셈 순서 문제(재귀)

```
rMatrixChain(i, j):        ▷ 행렬곱 A_i…A_j을 구하는 최소 비용을 구한다.
    if (i=j) return 0      ▷ 행렬이 하나뿐인 경우의 비용은 0
    min ← ∞
    for k ← i to j-1
        ❶ q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + p_{i-1}p_k p_j
          if (q < min) min ← q
    return min
```

✓ 엄청나게 많은 중복 호출

행렬 곱셈 순서 문제(DP)

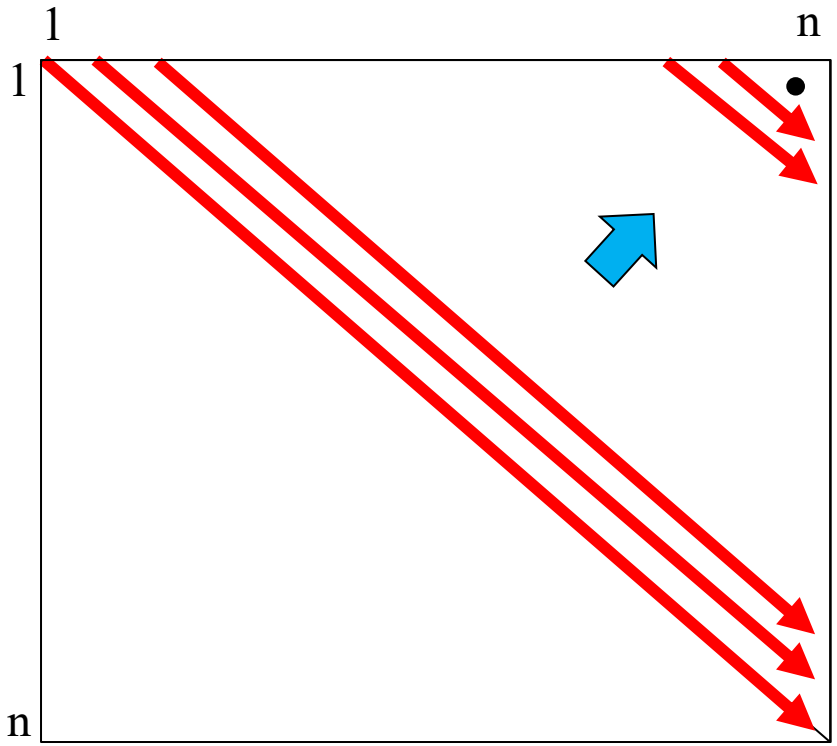
알고리즘 9-9 행렬 곱셈 순서 문제(동적 프로그래밍)

```
matrixChain(n):
    for i ← 1 to n
        m[i, i] ← 0          ▷ 행렬이 하나뿐인 경우의 비용은 0
    for r ← 1 to n−1         ▷ r: 문제의 크기를 결정하는 변수, 문제의 크기 = r+1
        for i ← 1 to n−r     ❶
            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]
```

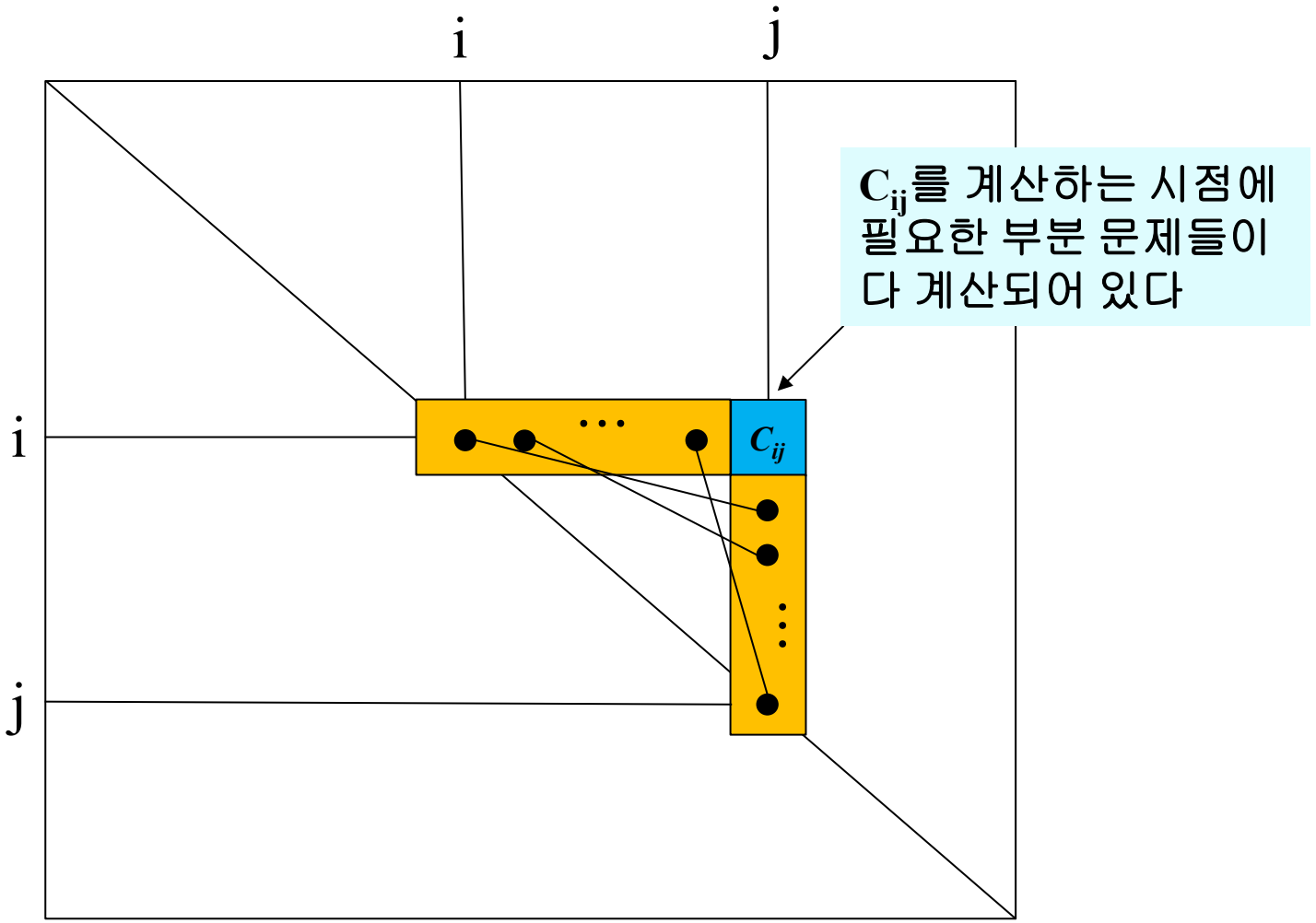
✓ 수행 시간: $\Theta(n^3)$

행렬 곱셈 순서 문제(DP)

[알고리즘 9-9]의 ❶



행렬 곱셈 순서 문제(DP)



행렬 곱셈 순서 문제(DP)

NOTE_ 알고리즘 rMatrixChain()의 수행 시간

[알고리즘 9–8]의 rMatrixChain() 알고리즘의 수행 시간은 ❶이 속한 for 루프가 좌우한다. 따라서 총 수행 시간은 이 for 루프 안의 한 장소인 ❶을 지나가는 총 횟수를 기준으로 삼을 수 있다. rMatrixChain()에서 곱하고자 하는 행렬의 총수가 n일 때 ❶을 지나가는 총 횟수는 다음과 같다.

$$P(n)=\begin{cases}0 & \text{if } n=1\\ \sum_{k=1}^{n-1}(P(k)+P(n-k)+1) & \text{if } n>1\end{cases}$$

$$=\begin{cases}0 & \text{if } n=1\\ 2\sum_{k=2}^{n-1}P(k)+n-1 & \text{if } n>1 \quad \leftarrow P(1)=0\end{cases}$$

$P(n)\geq c2^n$임을 증명해보자.

$$\begin{aligned}P(n)&=2\sum_{k=2}^{n-1}P(k)+n-1\\ &\geq 2\sum_{k=2}^{n-1}c2^k+n-1 \quad \leftarrow \text{귀납적 가정 이용}\\ &=2c(2^n-4)+n-1\\ &=2c2^n-8c+n-1\\ &\geq c2^n\end{aligned}$$

$c=\frac{1}{4}$이면 2 이상의 모든 n에 대해 성립한다. 따라서 알고리즘 rMatrixChain은 n에 대해 지수 함수적으로 증가한다. 즉, 수행 시간은 $\Omega(2^n)$이다. 매우 비효율적인 알고리즘임을 알 수 있다.

최장 공통 부분 순서(LCS) 문제

최장 공통 부분 순서(LCS) 문제

- 두 문자열에 공통적으로 들어있는 공통 부분순서 중 가장 긴 것을 찾는다.

부분 순서의 예

- <bcdb>는 문자열 <abcbdab>의 부분 순서다.

공통 부분 순서의 예

- <bca>는 문자열 <abcbdab>와 <bdcaba>의 공통 부분 순서다.

최장 공통 부분 순서(LCS) 문제

최적 부분 구조

- $x_m=y_n$이면 X_m과 Y_n의 LCS의 길이는 X_{m-1}과 Y_{n-1}의 LCS 길이보다 1이 크다. 즉, (m, n) 크기 문제의 해가 $(m-1, n-1)$ 문제의 해를 포함한다.
- $x_m \neq y_n$이면 X_m과 Y_n의 LCS 길이는 X_m과 Y_{n-1}의 LCS 길이와 X_{m-1}과 Y_n의 LCS 길이 중 큰 것과 같다. 즉, (m, n) 크기 문제의 해가 $(m, n-1)$ 크기 문제의 해와 $(m-1, n)$ 크기 문제의 해를 포함한다.

정리 9-1

$X_m=\langle x_1x_2\cdots x_m\rangle$과 $Y_n=\langle y_1y_2\cdots y_n\rangle$의 최장 공통 부분 순서 LCS의 길이가 k라 하자. LCS는 하나 이상 있을 수 있는데 이 중 하나를 $Z_k=\langle z_1z_2\cdots z_k\rangle$라 하자.

❶ $x_m=y_n$이면, $z_k=x_m=y_n$이고 Z_{k-1}은 X_{m-1}과 Y_{n-1}의 LCS다.

❷ $x_m \neq y_n$이고 $z_k \neq x_m$이면, Z_k는 X_{m-1}과 Y_n의 LCS다.

❸ $x_m \neq y_n$이고 $z_k \neq y_n$이면, Z_k는 X_m과 Y_{n-1}의 LCS다.

최장 공통 부분 순서(LCS) 문제(재귀)

알고리즘 9-10 최장 공통 부분 순서 길이(재귀)

```
LCS(m, n):
▷ 두 문자열 X_m과 Y_n의 LCS의 길이를 구한다.
    if (m = 0 or n = 0) return 0
    else if (x_m = y_n) return LCS(m−1, n−1)+1
    else return max(LCS(m−1, n), LCS(m, n−1))
```

최장 공통 부분 순서(LCS) 문제(재귀)

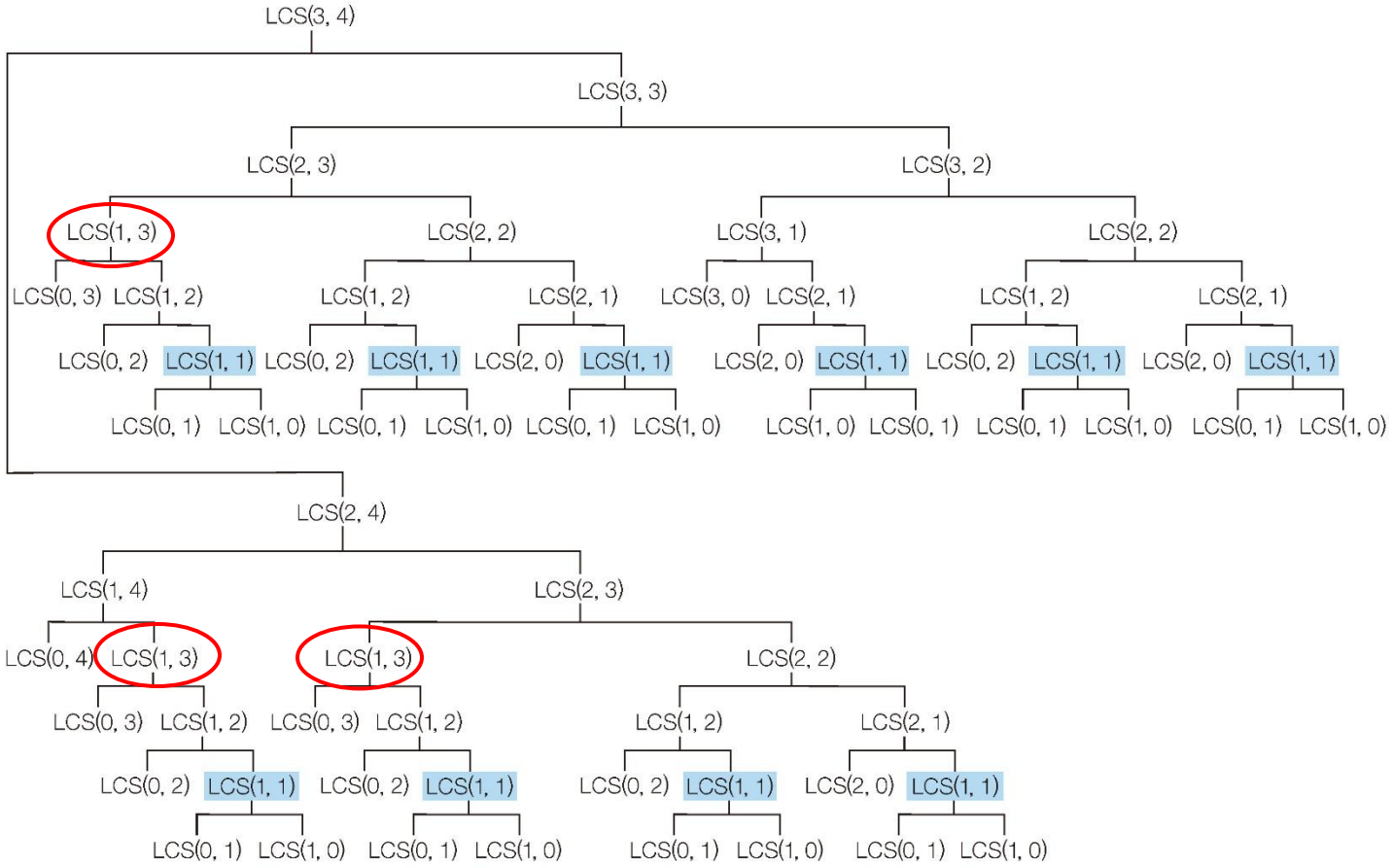


그림 9-9 LCS(3, 4)를 수행하는 과정의 재귀적 호출 관계를 나타내는 트리

표 9-4 LCS( )에서 문제 크기가 커짐에 따라 중복 호출이 증가하는 모습

수행되는 LCS( )	LCS(1, 1)의 중복 호출 횟수
LCS(2, 2)	2
LCS(2, 3)	3
LCS(3, 3)	6
LCS(3, 4)	10
LCS(4, 4)	20
LCS(4, 5)	35
LCS(5, 5)	70
LCS(5, 6)	126
LCS(6, 6)	252
LCS(6, 7)	462
LCS(7, 7)	924

최장 공통 부분 순서(LCS) 문제(DP)

알고리즘 9-11 최장 공통 부분 순서 길이(동적 프로그래밍)

```
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
①   for i ← 1 to m
        for j ← 1 to n
            if (x_i = y_i) 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]
```

✓ 수행 시간: $\Theta(mn)$

메모하기

피보나치 수열

알고리즘 9-12 피보나치 수열(메모하기)

```
▷ 배열 f[0...n-1]의 모든 원소는 0으로 초기화되어 있다.
▷ f[i]값이 0이면 fib(i)가 아직 한 번도 수행되지 않았음을 의미한다.
fib(n):
  ❶ if (f[n] ≠ 0) return f[n]
    else
        if (n = 1 or n = 2) f[n] ← 1
        else
            ❷ f[n] ← fib(n-1)+fib(n-2)
        return f[n]
```

참고 [알고리즘 9-1]의 피보나치 수(재귀)

```
fib(n):
    if (n=1 or n=2) return 1
  ❶ else return (fib(n-1)+fib(n-2))
```

메모하기

■ 돌 놓기 문제

알고리즘 9-13 돌 놓기 문제(메모하기 1)

▷ 아래와 같이 초기화해놓고 시작한다.

▷ $peb[i, p] = -\infty$, i=2, 3, ..., n, p=1, 2, 3, 4

▷ $peb[1, p] = w_{1p}$, p=1, 2, 3, 4

```
pebble(i, p):
❶ if (peb[i, p] ≠ −∞) return peb[i, p]
   else
        max ← −∞
        for every pattern q compatible to pattern p
            tmp ← pebble(i−1, q)
            if (tmp > max) max ← tmp
   ❷ peb[i, p] ← max + w_ip
        return peb[i, p]
```

참고 [알고리즘 9-6]의 돌 놓기 문제(재귀)

```
pebble(i, p):
▷ i열이 패턴 p로 놓일 때 최고 점수
▷ w_ip: i열이 패턴 p로 놓일 때 i열에 돌이 놓인 곳의 점수 합, p∈{1, 2, 3, 4}
    if (i=1) return w[1, p]
    else
        max ← −∞
        for q ← 1 to 4
            if (패턴 q가 패턴 p와 양립)
                tmp ← pebble(i−1, q)
                if (tmp > max) max ← tmp
        return (max + w_ip)
```

메모하기

■ 행렬 곱셈 순서 문제

알고리즘 9-15 행렬 곱셈 순서 문제(메모하기)

```
rMatrixChain(i, j):          ▷ 행렬곱 A_i…A_j를 구하는 최소 비용을 구한다.
❶ if(m[i, j] = ∞)
      if(i = j) m[i, j] = 0  ▷ 행렬이 하나뿐인 경우의 비용은 0
      else
          for k ← i to j-1
              q ← rMatrixChain(i, k) + rMatrixChain(k+1, j) + p_{i-1}p_k p_j
          if (q < m[i, j]) m[i, j] ← q
❷ return m[i, j]
```

참고 [알고리즘 9-8]의 행렬 곱셈 순서 문제(재귀)

```
rMatrixChain(i, j):          ▷ 행렬곱 A_i…A_j을 구하는 최소 비용을 구한다.
    if (i=j) 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_k p_j
        if (q < min) min ← q
    return min
```