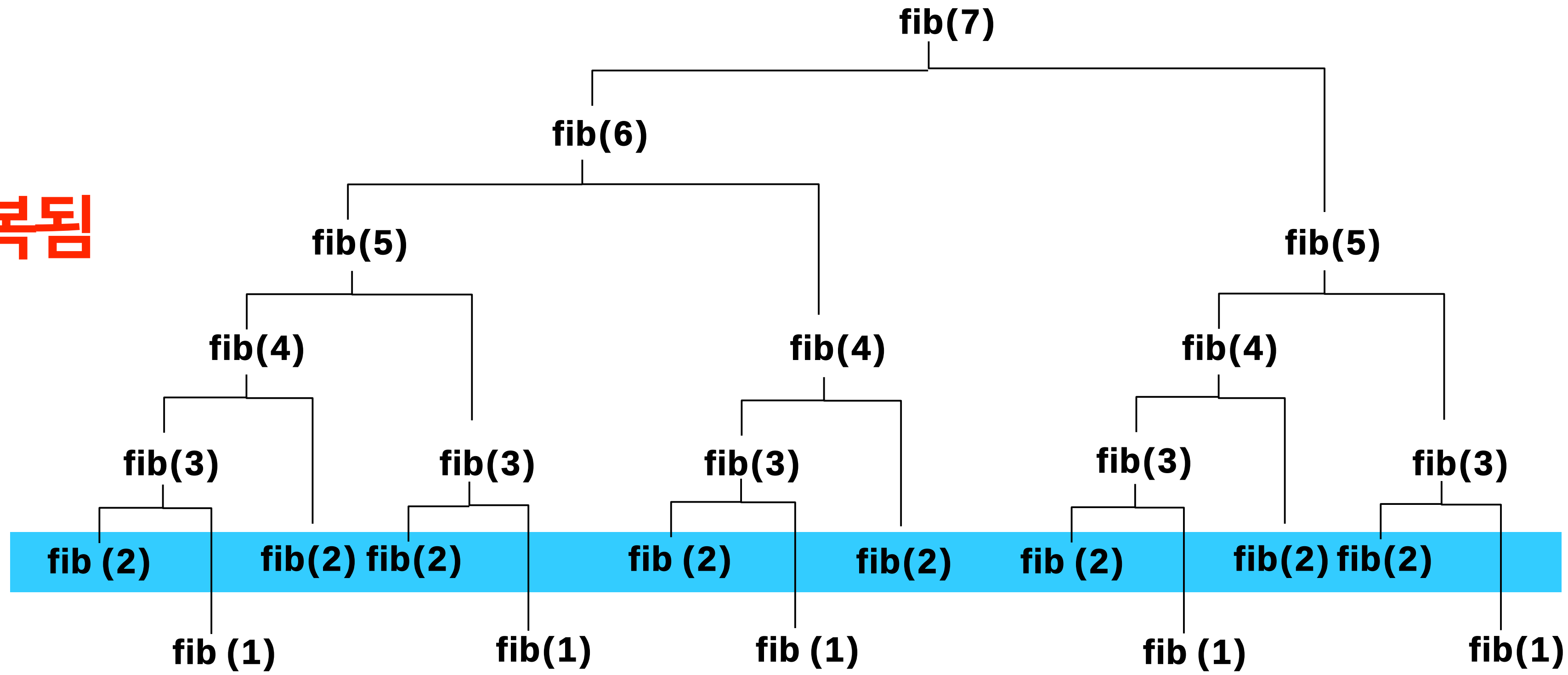
제5장
동적계획법
(Dynamic Programming)

Motivation

Fibonacci Numbers

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

많은 계산이 중복됨



Memoization

```
int fib(int n)
{
    if (n==1 || n==2)
        return 1;
    else if (f[n] > -1)  /* 배열 f가 -1으로 초기화되어 있다고 가정 */
        return f[n];     /* 즉 이미 계산된 값이라는 의미 */
    else {
        f[n] = fib(n-2) + fib(n-1);  /* 중간 계산 결과를 caching */
        return f[n];
    }
}
```

중간 계산 결과를 caching
함으로써 중복 계산을 피함

	1	2	3	4	5	6	7	8	9
f	1	1	2	3	5	8	13	21	34

cache

Dynamic Programming

```
int fib(int n)
{
    f[1] = f[2] = 1;
    for (int i=3; i<=n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

	1	2	3	4	5	6	7	8	9
f	1	1	2	3	5	8	13	21	34

bottom-up 방식

bottom-up 방식으로
중복 계산을 피함

Memoization vs. Dynamic Programming

- 순환식의 값을 계산하는 기법들이다.
- 둘 다 동적계획법의 일종으로 보기도 한다.
- Momoization은 top-down방식이며, 실제로 필요한 subproblem만을 푼다.
- 동적계획법은 bottom-up 방식이며, recursion에 수반되는 overhead가 없다.

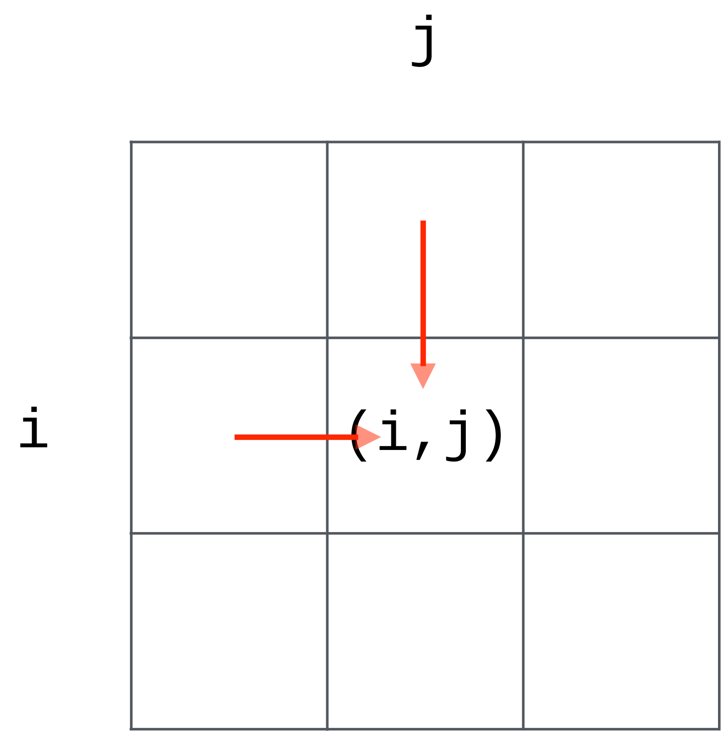
Basic Example

행렬 경로 문제

- 정수들이 저장된 n×n 행렬의 좌상단에서 우하단까지 이동한다. 단 오른쪽이나 아래쪽 방향으로만 이동할 수 있다
- 방문한 칸에 있는 정수들의 합이 최소화되도록 하라.

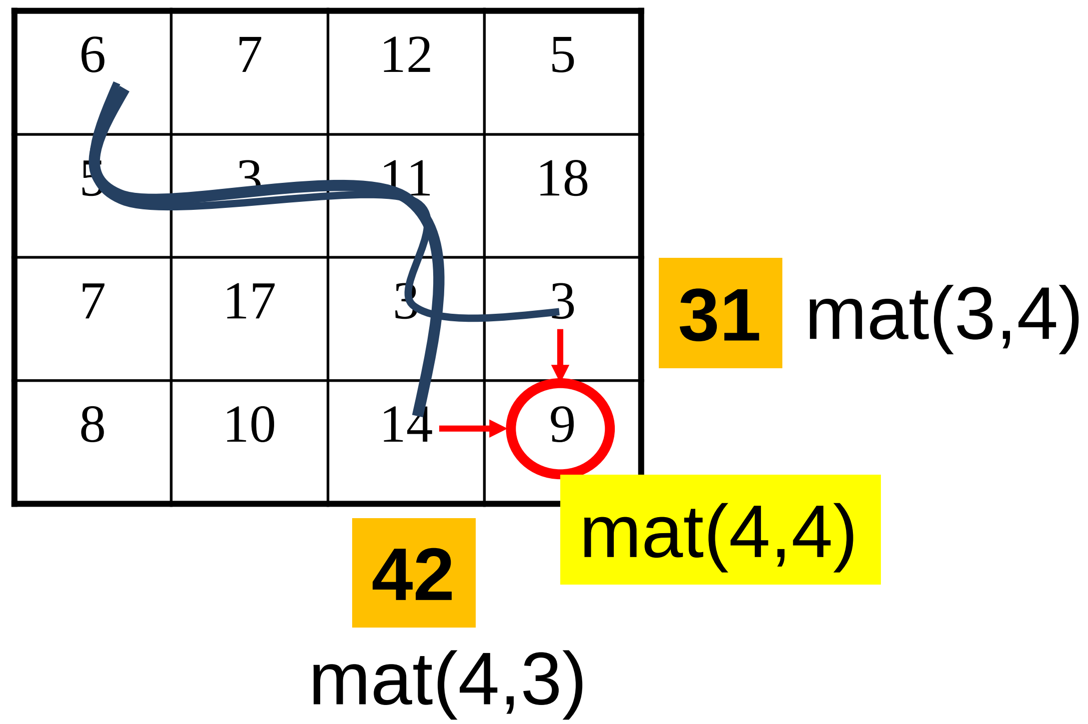
	1	2	3	4
1	6	7	12	5
2	5	3	11	18
3	7	17	3	3
4	8	10	14	9

Key Observation



(i,j)에 도달하기 위해서는 (i,j-1) 혹은 (i-1,j)를 거쳐야 한다.

또한 (i,j-1) 혹은 (i-1,j)까지는 최선의 방법으로 이동해야 한다.



$$mat(4,4) = mat(3,4) + M[4][4] = 40$$

int mat (int i, int j)

(1,1)에서 (i,j)까지의 최저점수를 구하는 함수

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

mat(3,3)

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

mat(4,3) = Min(mat(4,2),mat(3,3)) + M[4][3]

int matrixPath(int i, int j)
(1,1)에서 (i,j)까지의 최저점수를 구하는 함수

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

mat(2,4)

mat(3,3) mat(3,4)

mat(3,4) = Min(mat(3,3),mat(2,4)) + M[3][4]

int matrixPath(int i, int j)
(1,1)에서 (i,j)까지의 최저점수를 구하는 함수

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

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

mat(1,3) = mat(1,2) + M[1][3]

int matrixPath(int i, int j)
(1,1)에서 (i,j)까지의 최저점수를 구하는 함수

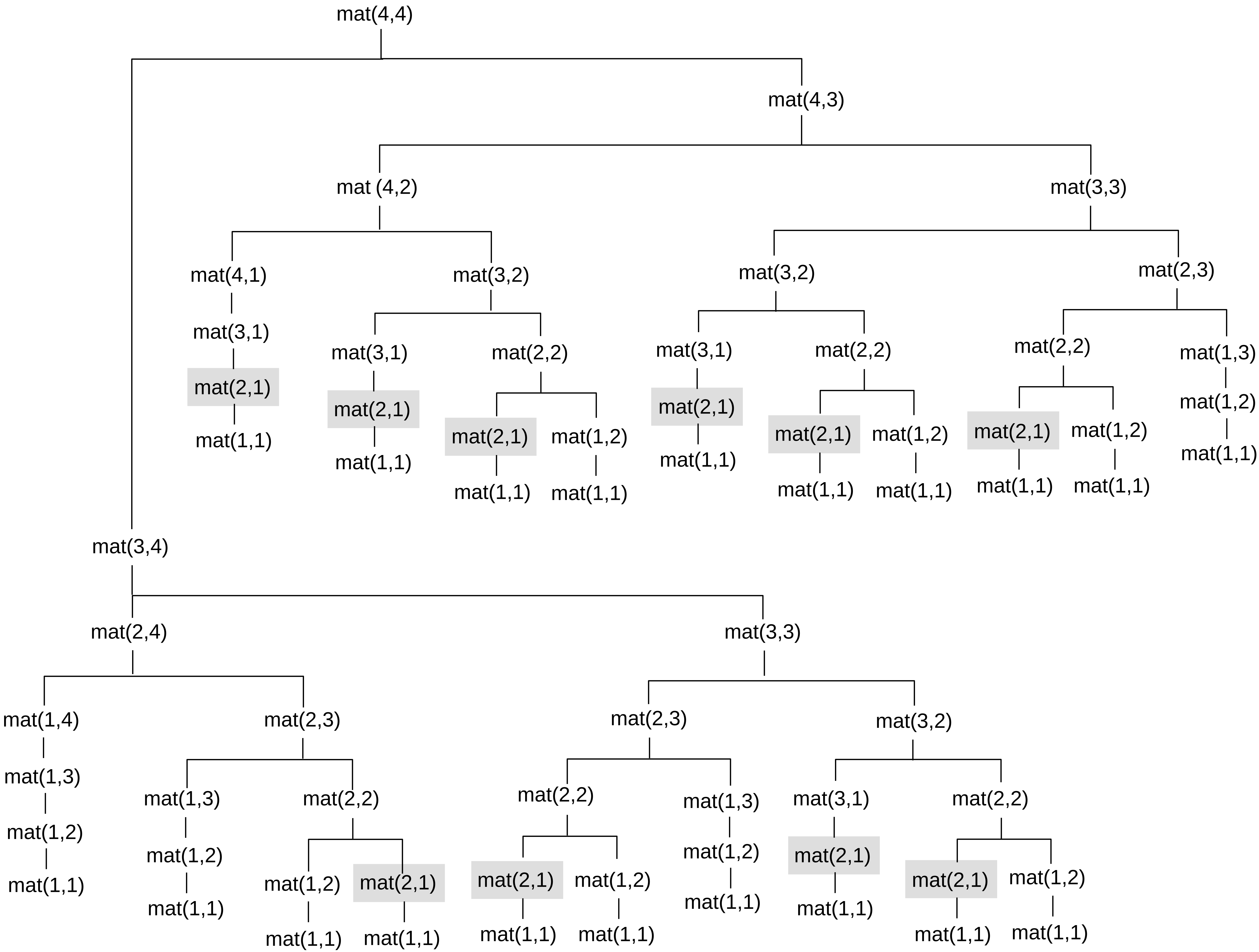
	6	7	12	5
mat(2,1)	5	3	11	18
mat(3,1)	7	17	3	3
	8	10	14	9

mat(3,1) = mat(2,1) + M[3][1]

int matrixPath(int i, int j)
(1,1)에서 (i,j)까지의 최저점수를 구하는 함수

Recursive Algorithm

```
int mat(int i, int j)
{
    if (i == 1 && j == 1)
        return m[i][j];
    else if (i == 1)
        return mat(1, j-1) + m[i][j];
    else if (j == 1)
        return mat(i-1, 1) + m[i][j];
    else
        return Math.min(mat(i-1, j), mat(i, j-1)) + m[i][j];
}
```


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)

Memoization

```
int mat(int i, int j)
{
    if (L[i][j] != -1) return L[i][j];
    if (i == 1 && j == 1)
        L[i][j] = m[i][j];
    else if (i == 1)
        L[i][j] = mat(1, j-1) + m[i][j];
    else if (j == 1)
        L[i][j] = mat(i-1, 1) + m[i][j];
    else
        L[i][j] = Math.min(mat(i-1, j), mat(i, j-1)) + m[i][j];
    return L[i][j];
}
```

순환식 (Dynamic Programming)

- $L[i, j]$: (1, 1)에서 (i, j)까지 이르는 최소합

$$L[i,j] = \begin{cases} m_{ij} & \text{if } i = 1 \text{ and } j = 1; \\ L[i-1,j] + m_{ij} & \text{if } j = 1; \\ L[i,j-1] + m_{ij} & \text{if } i = 1; \\ \min(L[i-1,j], L[i,j-1]) + m_{ij} & \text{otherwise.} \end{cases}$$

Bottom-Up

m

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

L	1	2	3	4
1	6	13	25	30
2	11	14	25	43
3	18	31	28	31
4	26	36	42	40

순서로 계산하면 필요한 값이 항상 먼저 계산됨

Bottom-Up

```
int mat()
{
   for  (int i=1; i<=n;  i++)  {
         for (int j=1;  j<=n; j++) {
             if (i==1 && j==1)
                 L[i][j] = m[1][1];
             else if (i==1)
                 L[i][j] = m[i][j] + L[i][j-1];
             else if (j==1)
                 L[i][j] = m[i][j] + L[i-1][j];
             else
                 L[i][j] = m[i][j] + Math.min(L[i-1][j],L[i][j-1]);
        }
    }
    return L[n][n];
}
```

시간복잡도: $O(n^2)$

Common Trick

```
/* initialise L with L[0][j]=L[i][0]=∞ for all i and j */

int mat()
{
   for  (int  i=1;  i<=n;   i++)  {
          for  (int  j=1;   j<=n; j++)  {
             if (i==1 && j==1)
                 L[i][j] = m[1][1];
             else
                 L[i][j] = m[i][j] +  Math.min(L[i-1][j],L[i][j-1]);
      }
    }
    return L[n][n];
}
```

시간복잡도: O(n^2)

Optimal Substructure

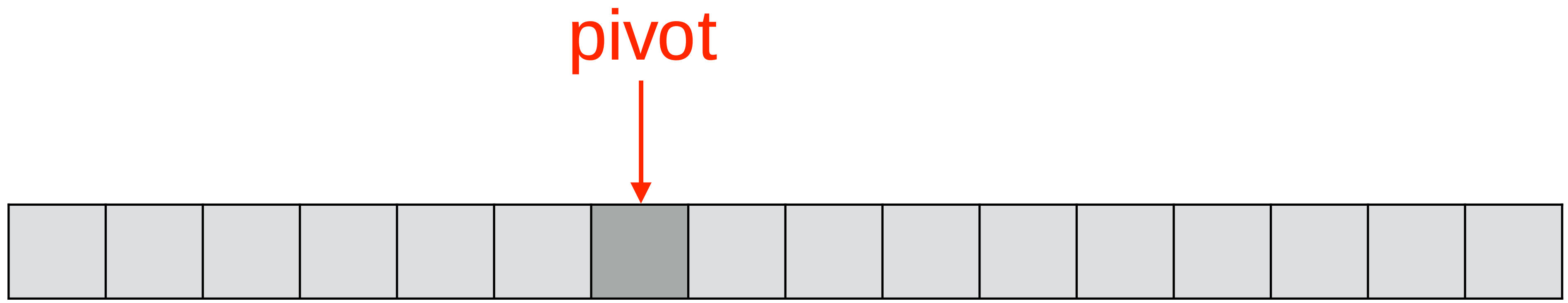
동적계획법

1. 일반적으로 최적화문제(optimisation problem) 혹은 카운팅(counting) 문제에 적용됨
2. 주어진 문제에 대한 순환식(recurrence equation)을 정의한다.
3. 순환식을 memoization 혹은 bottom-up 방식으로 푼다.

- subproblem들을 풀어서 원래 문제를 푸는 방식. 그런 의미에서 분할정복법과 공통성이 있음
- 분할정복법에서는 분할된 문제들이 서로 disjoint하지만 동적계획법에서는 그렇지 않음
- 즉 서로 overlapping하는 subproblem들을 해결함으로써 원래 문제를 해결

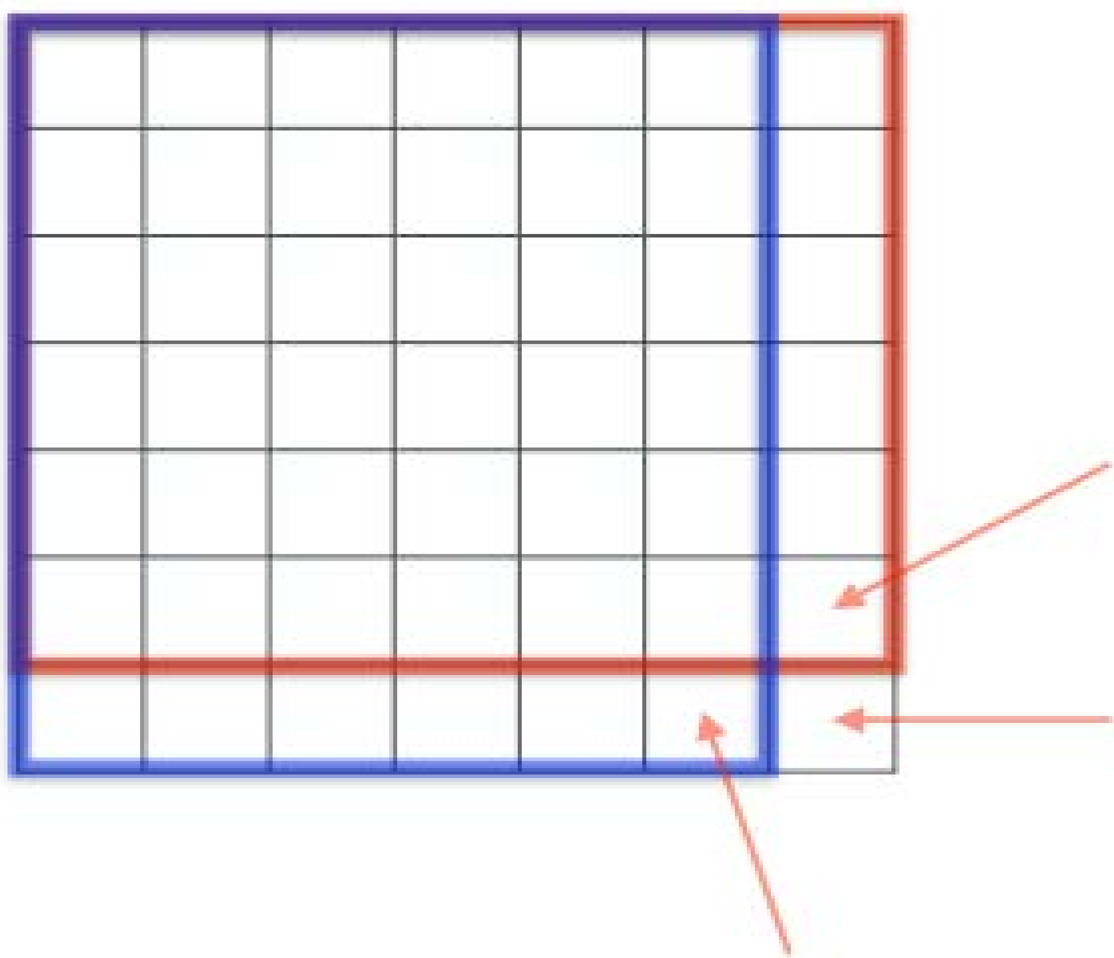
분할정복법 vs. 동적계획법

quicksort의 경우



pivot을 기준으로 분할된
두 subproblem은 서로 disjoint하다.

행렬경로문제의 경우



② 여기까지 오는 최적 해와

① 여기까지 오는 최적 해를 구하기 위해서

③ 여기까지 오는 최적 해를 구한다.

④ 하지만 ②번 해와 ③번 해는 disjoint하지 않다.

Optimal Substructure

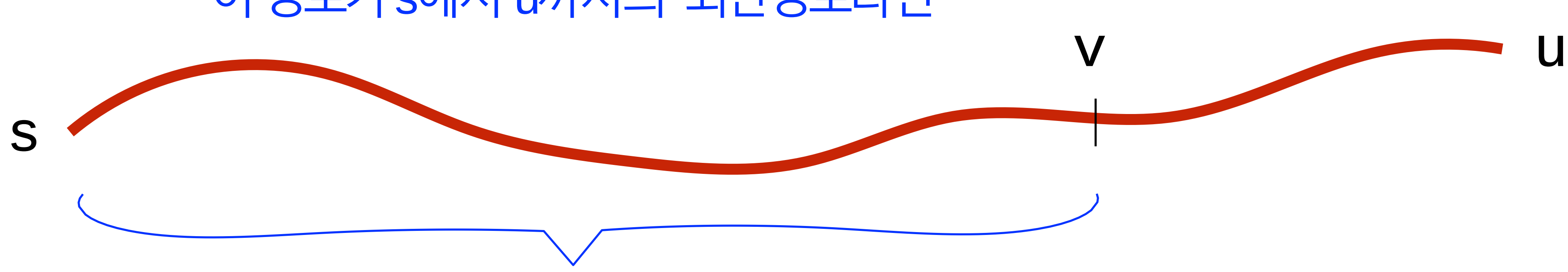
- 어떤 문제의 최적해가 그것의 subproblem들의 최적해로부터 효율적으로 구해질 수 있을 때 그 문제는 optimal substructure를 가진다고 말한다.

  (A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.)

- 분할정복법, 탐욕적기법, 동적계획법은 모두 문제가 가진 이런 특성을 이용한다.

Optimal Substructure를 확인하는 질문

- “최적해의 일부분이 그부분에 대한 최적해인가?”
- 최단경로(shortest-path) 문제



- 순환식은 optimal substructure를 표현한다.

$$d[u] = \min_{v \text{ adjacent to } u} (d[v] + w(v,u))$$

u까지 가는 최단경로의 길이는

u에 인접한 모든 정점 v에 대해서 v까지 가는 최단경로의 길이 더하기 에지 (v,u)의 가중치의 합의 최소값이다.

Longest Common Subsequence

Longest Common Subsequence(LCS)

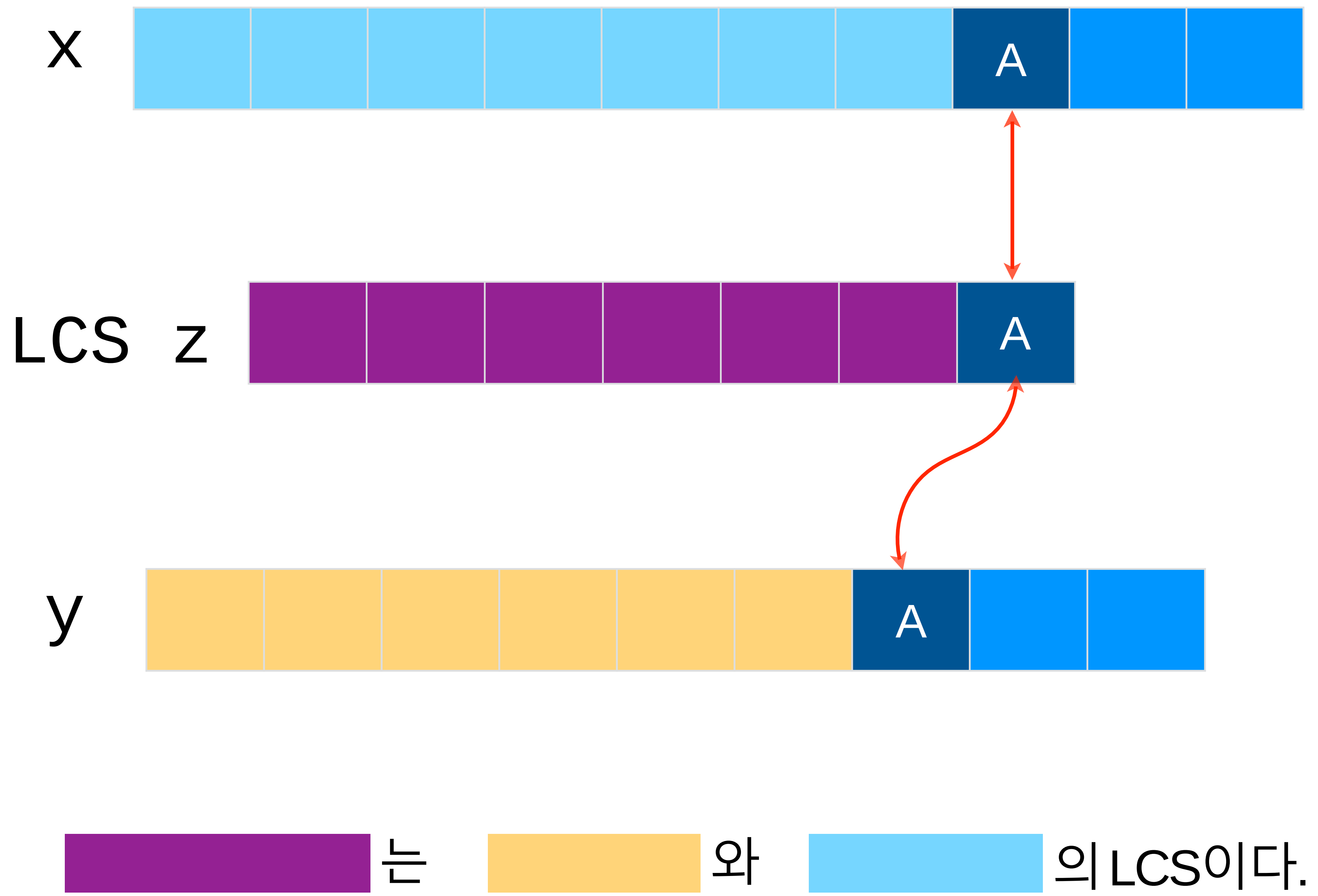
- <bcdb>는 문자열 <abcbdab>의 subsequence이다.
- <bca>는 문자열 <abcbdab>와 <bdcaba>의 common subsequence 이다.

- Longest common subsequence(LCS)
  - common subsequence들 중 가장 긴 것
  - <bcba>는 <abcbdab>와 <bdcaba>의 LCS이다

Brute Force

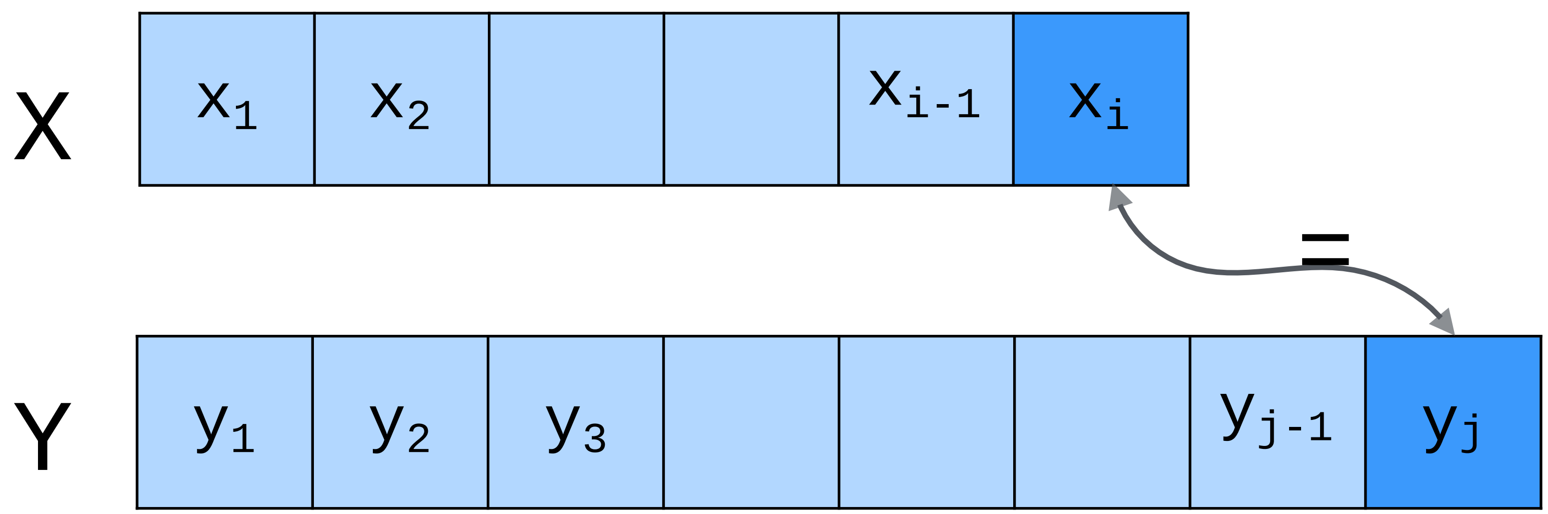
- 문자열 x의 모든 subsequence에 대해서 그것이 y의 subsequence가 되는지 검사한다.
- $|x|=m, |y|=n$
- x의 subsequence의 개수 = 2^m
- 각각이 y의 subsequence인지 검사: $O(n)$시간
- 시간복잡도 $O(n2^m)$

Optimal Substructure



순환식

- $LCS(i, j)$: 문자열 $X = \langle x_1 x_2 \cdots x_i \rangle$와 $Y = \langle y_1 y_2 \cdots y_j \rangle$의 LCS의 길이

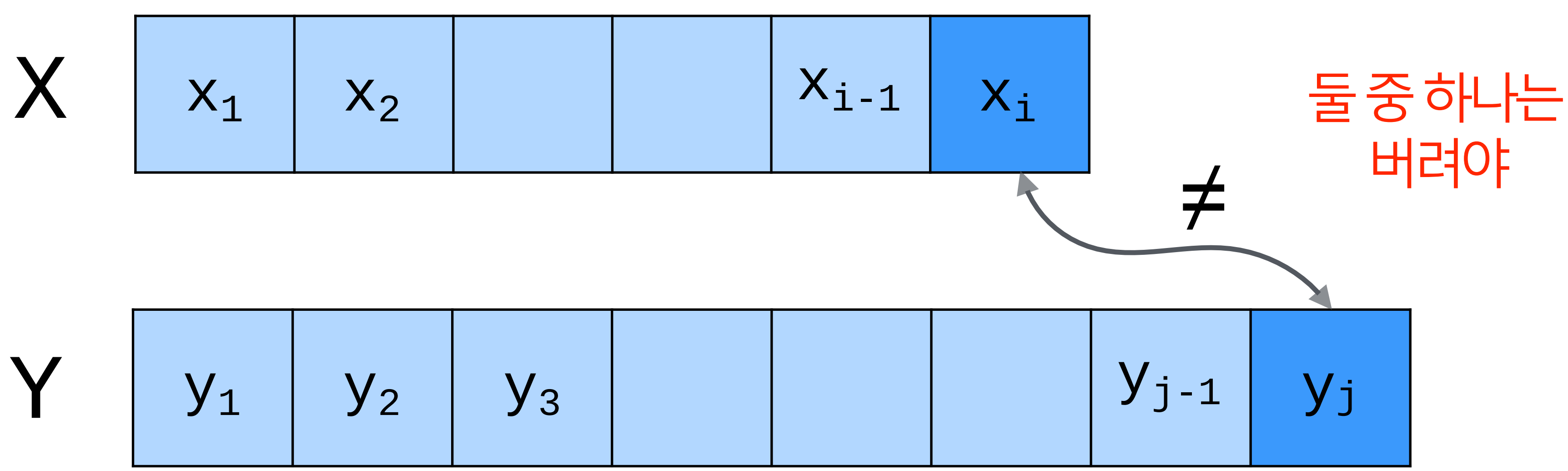


- 경우1: $x_i = y_j$

$$\mathrm{LCS}(i, j) = LCS(i - 1, j - 1) + 1$$

순환식

- $LCS(i, j)$: 문자열 $X = \langle x_1 x_2 \cdots x_i \rangle$와 $Y = \langle y_1 y_2 \cdots y_j \rangle$의 LCS의 길이



- 경우2: $x_i \mathrel{!=} y_j$

$$LCS(i, j) = \max(LCS(i-1, j), LCS(i, j-1))$$

X=“abcbdab" Y=“bdcaba”

LCS(X, Y, 7, 6)
X_7과 Y_6가 다르기 때문에
LCS(X,Y,6,6)와 LCS(X,Y,7,5) 중의 큰 값

X=“abcbda" Y=“bdcaba”

X=“abcbdab" Y=“bdcab”

X=“abcbda" Y=“bdcaba”

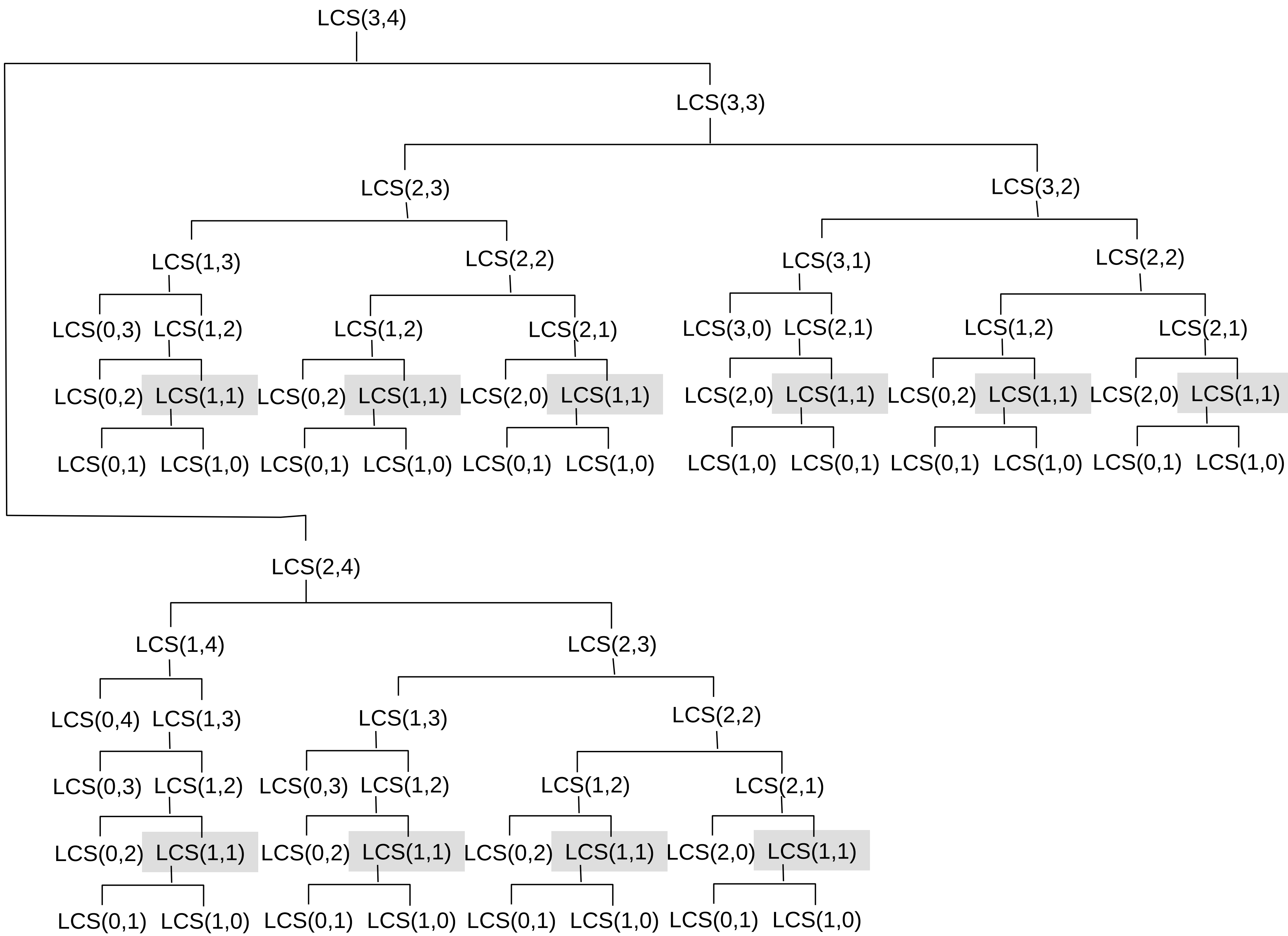
LCS(X, Y, 6, 6)
X_6과 Y_6가 같기 때문에
LCS(X,Y,5,5) + 1

X=“abcbd" Y=“bdcab”

```
int LCS(char*X, char *Y, m, n)
▷ 두 문자열 X (길이 m)과 Y (길이 n) 의 LCS 길이 구하기
{
      if (m = 0 or n = 0)
            then return 0;
      else if (X[m-1]= y[n-1])
            then return LCS(X,Y,m-1, n-1) + 1;
      else
            return max(LCS(X,Y,m-1, n), LCS(X,Y,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)

순환식

- $L[i, j]$: 문자열 $X = < x_1 x_2 \cdots x_i >$와 $Y = < y_1 y_2 \cdots y_j >$의 LCS의 길이

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0; \\ L[i-1, j-1] + 1 & \text{if } x_i = y_j; \\ \max(L[i-1, j], L[i, j-1]) & \text{otherwise.} \end{cases}$$

예

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

X = ABCBDAB
Y = BDCABA

동적계획법

```
int lcs(int m, int n) /* m: length of X, n: length of Y */
{
    for (int i=0; i<=m; i++)
        c[i][0] = 0;
    for (int j=0; j<=n; j++)
        c[0][j] = 0;
    for (int i=0; i<=m; i++) {
        for (int j = 0; j <= n; j++) {
            if (x[i] == y[j])
                c[i][j] = c[i - 1][j - 1] + 1;
            else
                c[i][j] = Math.max(c[i - 1][j], c[i][j - 1]);
        }
    }
    return c[m][n];
}
```

시간복잡도: $\Theta(mn)$

Maximum Sum Interval

Maximum Sum Interval

- N 개의 정수 $a_1, a_2, . . . , a_N$이 주어진다.
  이 중 하나 혹은 그 이상의 연속 된 정수들을 더하여 만들 수 있는 최대값은?

Brute Force Algorithm

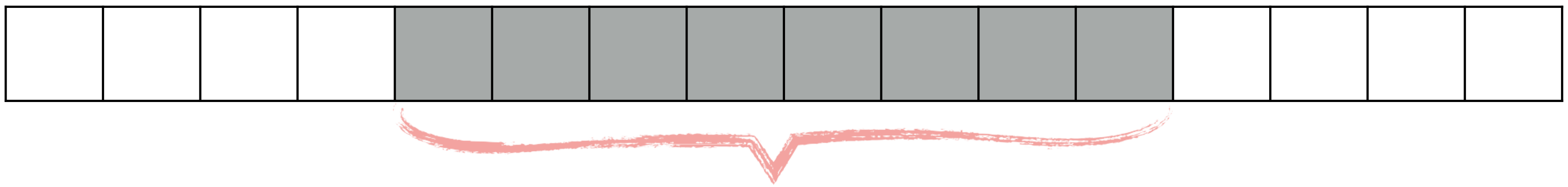
```
int max_sum(int *A, int n){
  int max = A[0];
  for(i=0; i < n ; i++ ) {
    int local_max = A[i];
    int sum = A[i];
    for(j=i+1; j < n ; j++ ) { //A[i]부터 시작하는 최대값을 구한다.
      sum = sum + A[j];
      if ( local_max < sum ) local_max = sum;
    }
    if ( max < local_max ) max = local_max;
  }
  return max;
}
```

시간복잡도 O(n^2)

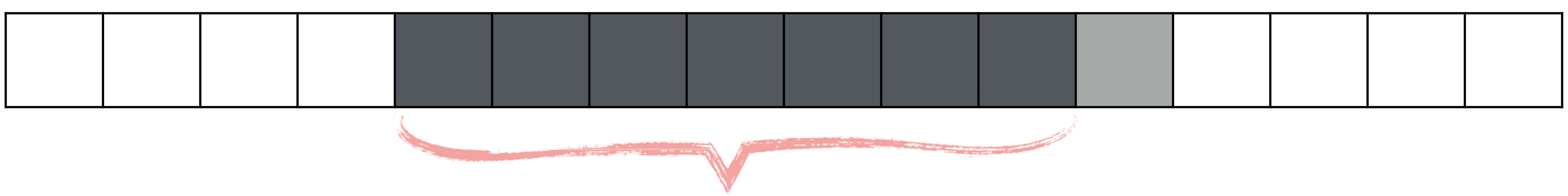
For your exercise...

시간복잡도 O(nlogn)

Optimal Substructure



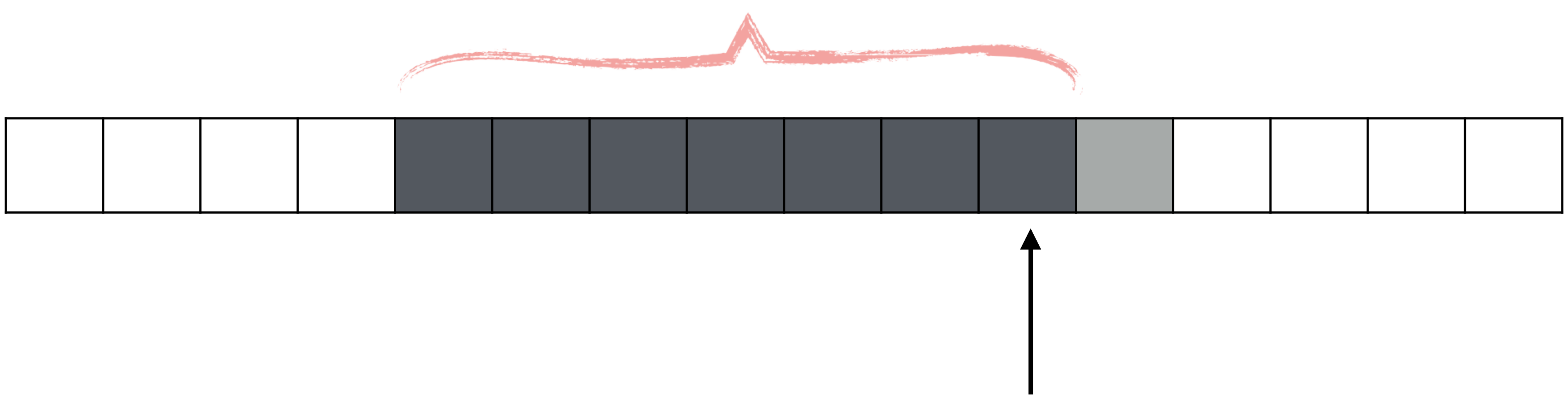
가령 이것이 합이 최대가 되는 구간이라고 가정해보자.



그렇다면 최적 구간의 일부인 이 구간의 정체는?

Optimal Substructure

그렇다면 최적 구간의 일부인 이 구간의 정체는?



끝점이 여기인 구간들 중에 합이 최대인 구간이다.

순환식

- maxEndsAt[*i*]: 끝점이 *i* 인 구간들 중 최대합

$$\text{maxEndsAt}[i] = \begin{cases} A[1] & i = 1; \\ \max(\text{maxEndsAt}[i-1] + A[i], A[i]) & i > 1. \end{cases}$$

- 최대합 = selection(maxEndsAt,1) where $i = 1, 2, \ldots, n$

Maximum Sum Interval

```
int maxSumInterval(int A[]) {     /* assume A[1],…,A[n]  */
    int maxEndsAt = A[1];
    int max = maxEndsAt[1];       /* 하나도 선택하지 않아도 되면 0으로 초기화 */
    for (int i=2; i<=n; i++) {
        maxEndsAt[i] = Math.max(maxEndsAt[i-1] + A[i], A[i]);
        if (maxEndsAt[i]> max)
            max = maxEndsAt[i];
    }
    return max;
}
```

시간복잡도 O(n)