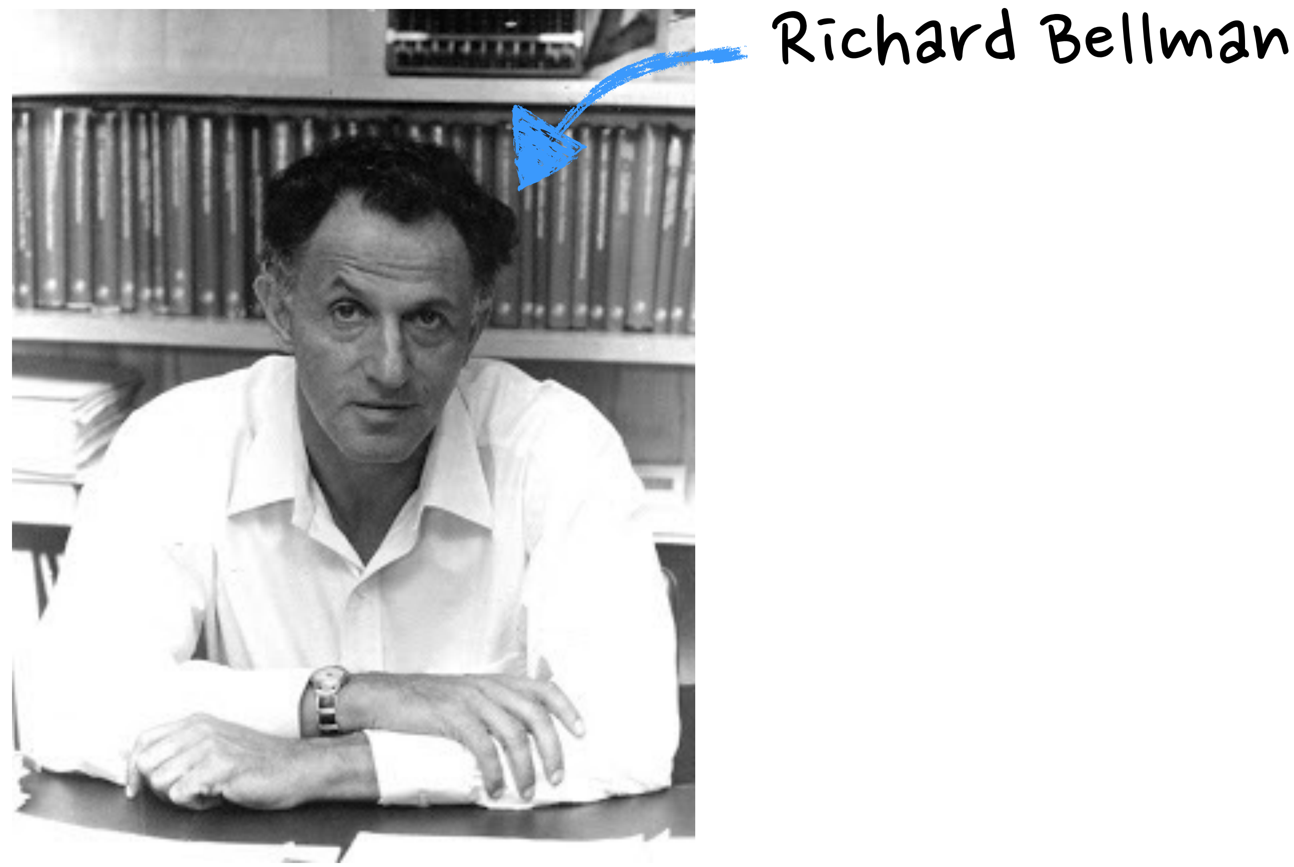


제7장

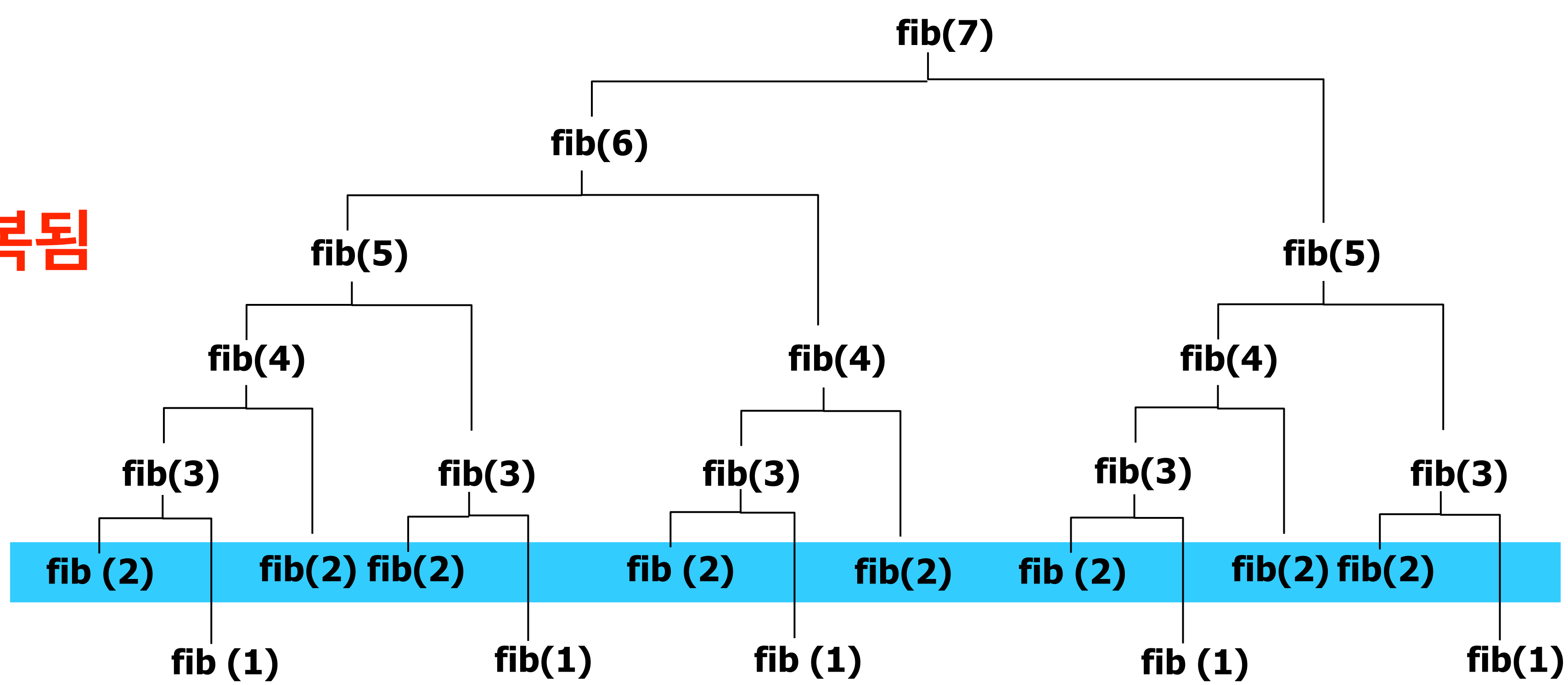
동적계획법 (Dynamic Programming)

Motivation

Fibonacci Numbers

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

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	5	6	7	8	9	10
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	5	6	7	8	9	10
f	1	1	2	3	5	8	13	21	34

bottom-up 방식

bottom-up 방식으로
중복 계산을 피함

이항 계수(Binomial Coefficient)

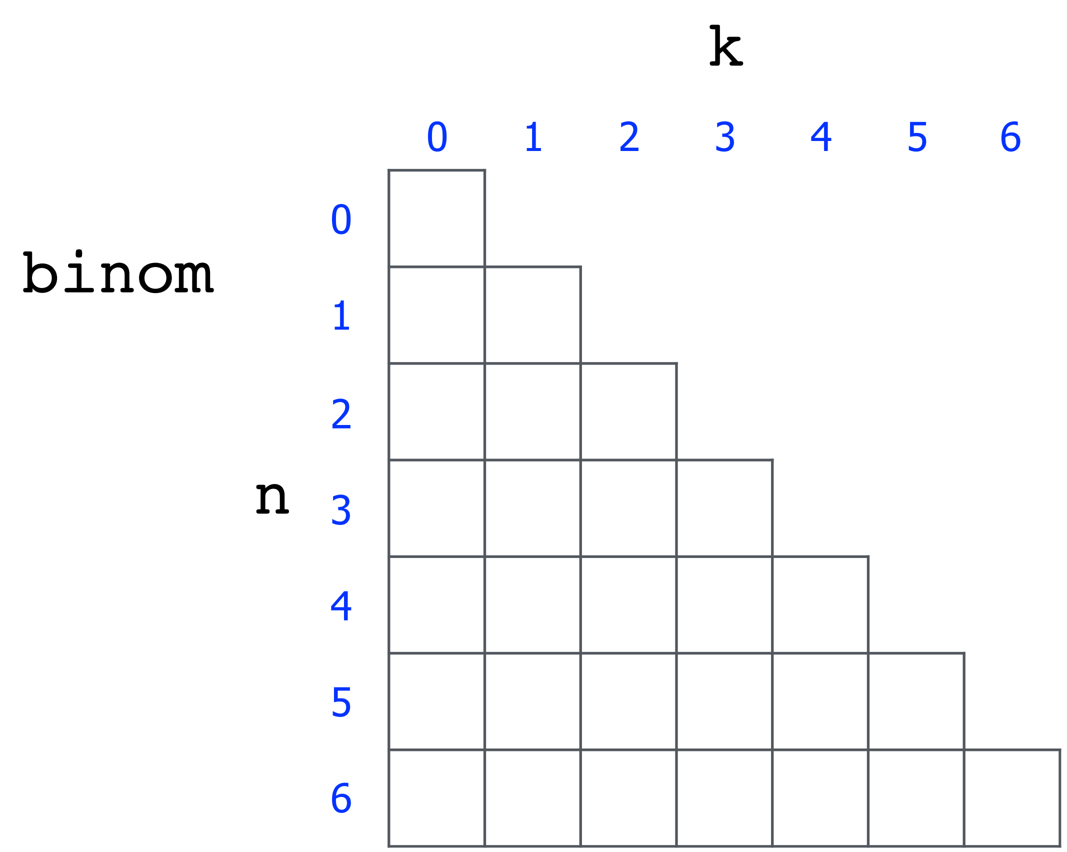
$$\binom{n}{k} = \begin{cases} 1 & \text{if } n = k \text{ or } k = 0; \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise.} \end{cases}$$

```
int binomial(int n, int k)
{
    if (n == k || k == 0)
        return 1;
    else
        return binomial(n - 1, k) + binomial(n - 1, k - 1);
}
```

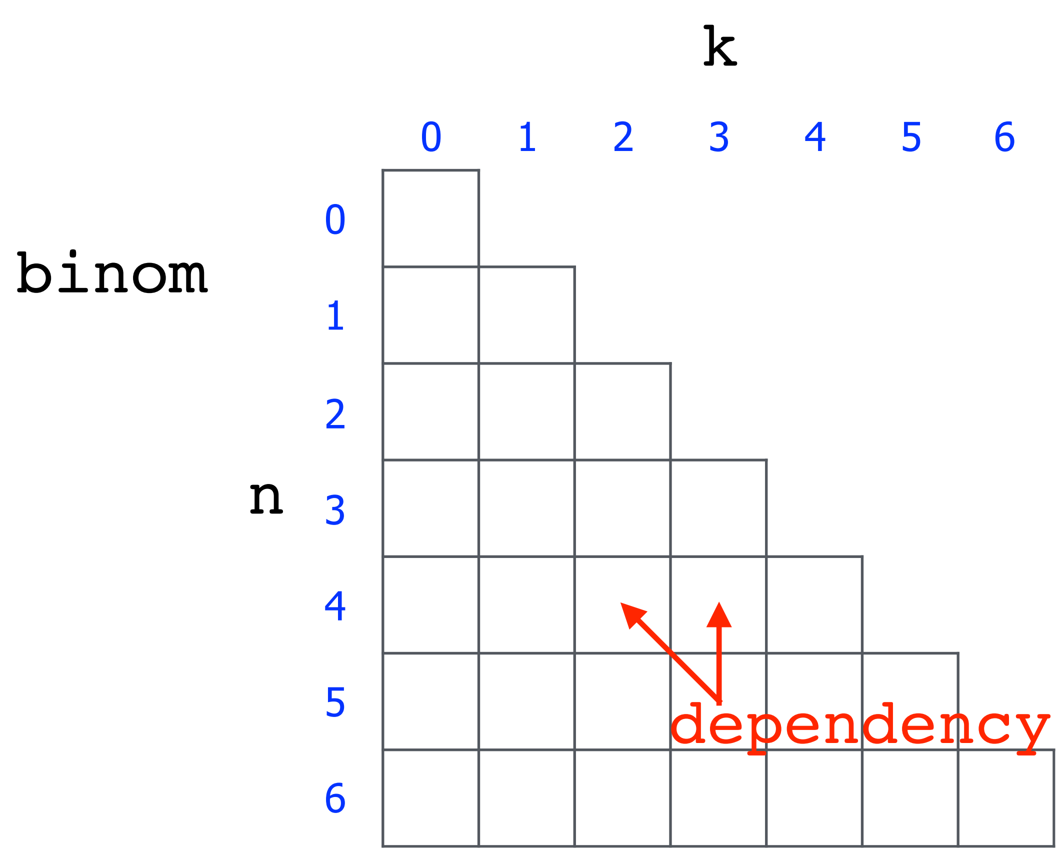
역시 많은 계산이 중복됨

Memoization

```
int binomial(int n, int k)
{
    if (n == k || k == 0)
        return 1;
    else if (binom[n][k] > -1) /* 배열 binom이 -1로 초기화되어 있다고 가정 */
        return binom[n][k];
    else  {
        binom[n][k] = binomial(n-1, k) + binomial(n-1, k-1);
        return binom[n][k];
    }
}
```



```
int binomial(int n, int k)
{
    for (int i=0; i<=n; i++) {
        for (int j=0; j<=k && j<=i; j++)  {
            if (k==0 || n==k)
                binom[i][j] = 1;
            else
                binom[i][j] = binom[i-1][j-1] + binom[i-1][j];
        }
    }
    return binom[n][k];
}
```



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.

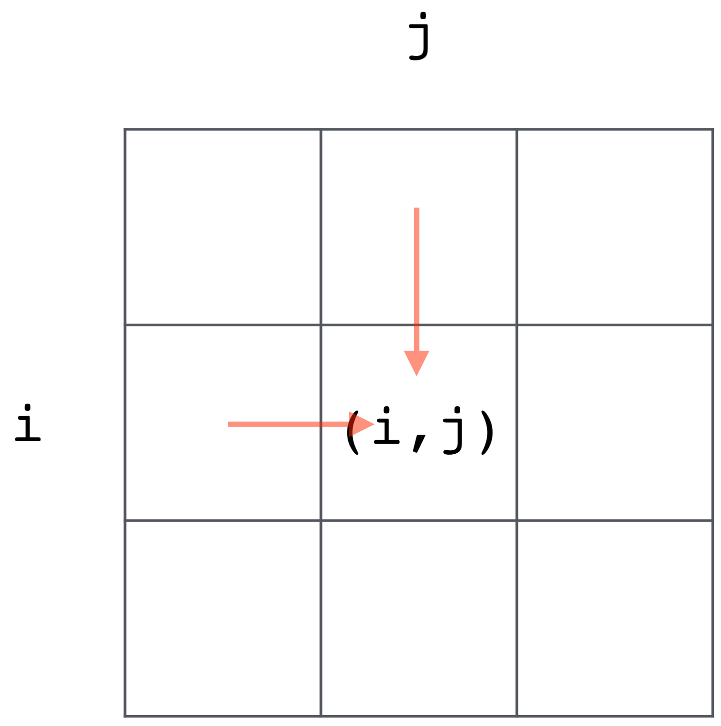
Basic Example

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.

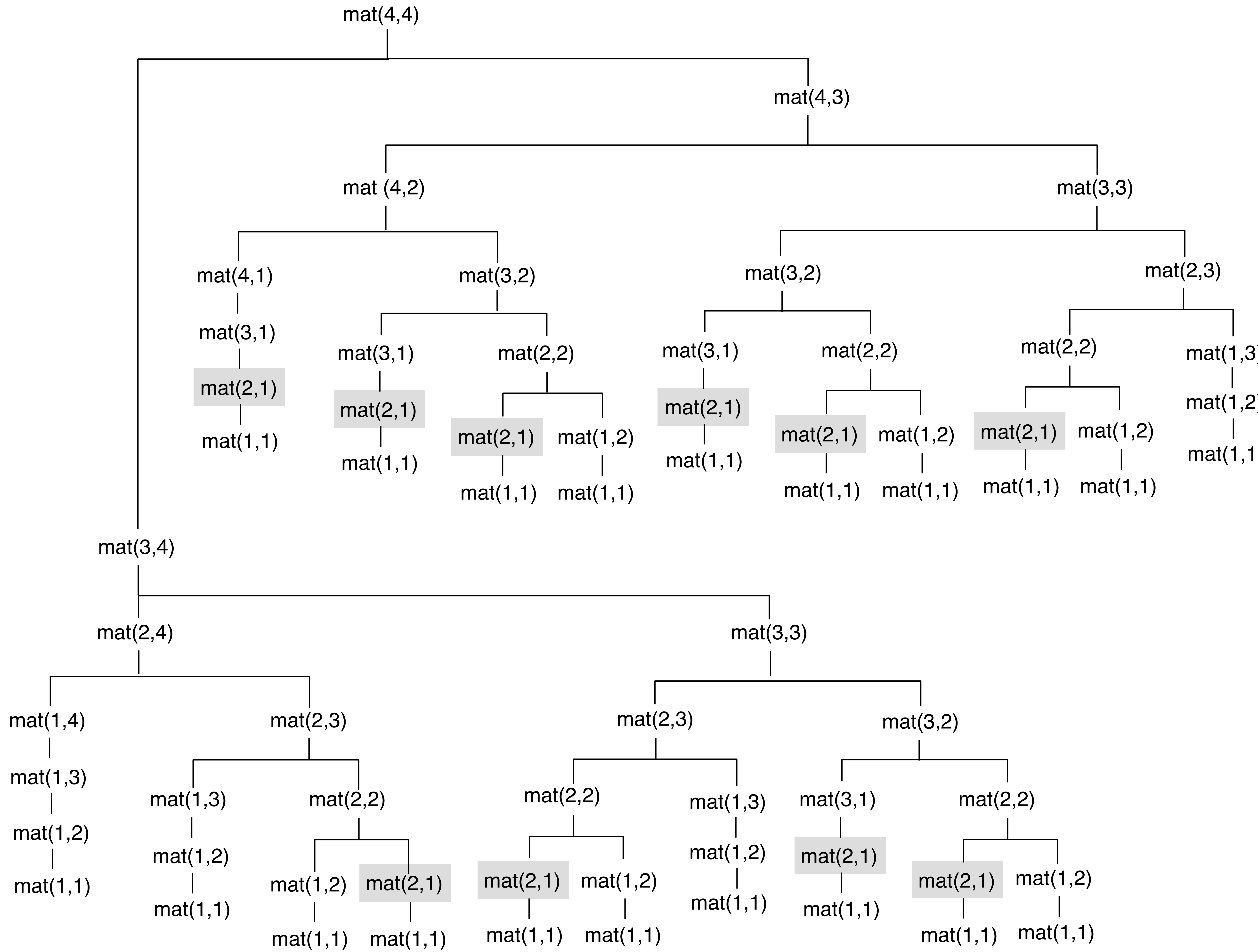
순환식

- $L[i,j]$: $(1,1)$에서 (i,j)까지 이르는 최소합

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

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];
}
```

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	16	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: $O(n^2)$

경로 구하기

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

P

-	←	←	←
↑	←	←	←
↑	↑	↑	←
↑	←	↑	↑

경로 구하기

```
/* 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];
                P[i][j] = '-';
            else  {
                if (L[i-1][j]<L[i][j-1]) {
                    L[i][j] = m[i][j] + L[i-1][j];
                    P[i][j] = '↑';
              }
              else {
                    L[i][j] = m[i][j] + L[i][j-1];
                    P[i][j] = '←';
              }
            }
        }
    }
    return L[n][n];
}
```

시간복잡도: $O(n^2)$

경로 구하기

```
void printPath()
{
    int i = n, j = n;
    while (P[i][j] != ‘-‘) {
        print(i + “ “ + j);
        if (P[i][j] == ‘←’)
            j = j-1;
        else
            i = i-1;
    }
    print(i + “ “ + j);
}
```

경로 구하기

```
void printPathRecursive(int i, int j)
{
    if (i==1 && j==1)
        print(1 + “ “ + 1);
    else {
        if (P[i][j] == ‘←’)
            printPathRecursive(i, j-1);
        else
            printPathRecursive(i-1, j);
        print(i + “ “ + j);
    }
}
```

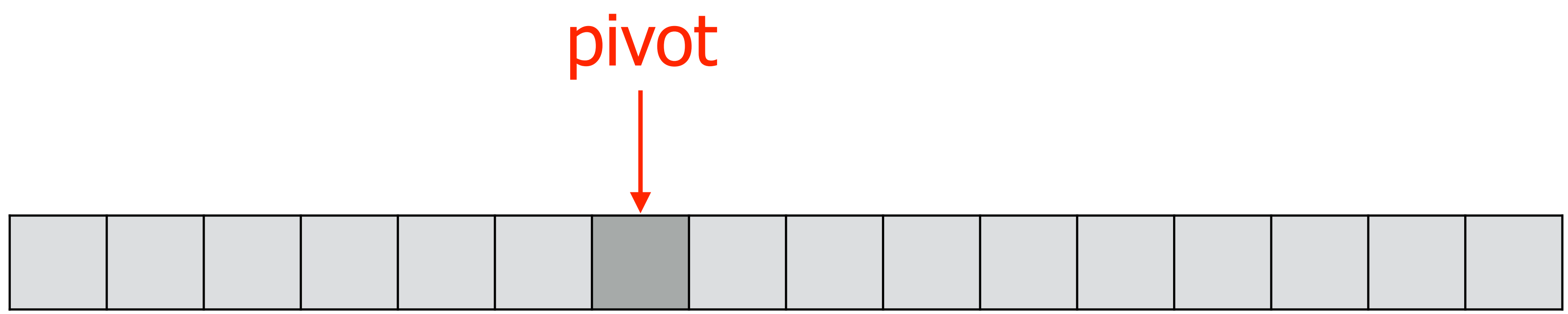
Optimal Substructure

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

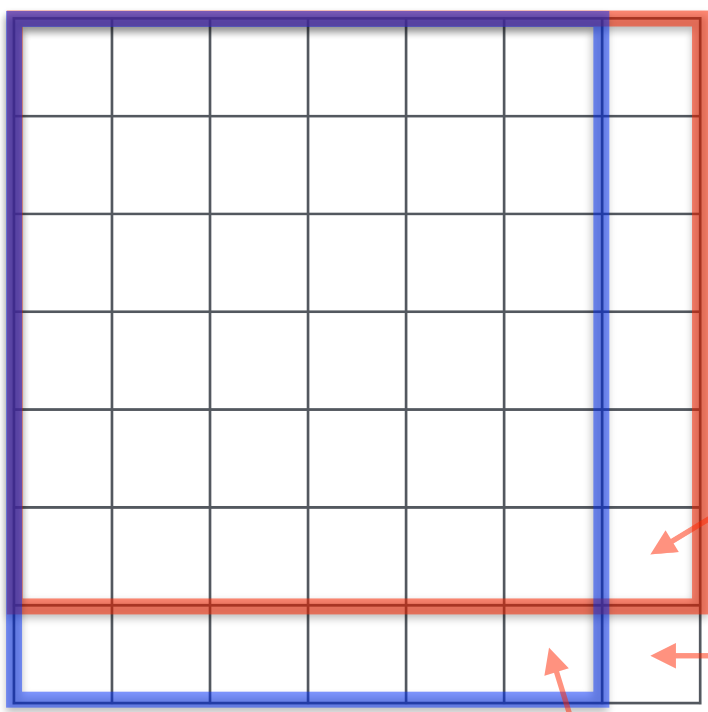
quicksort의 경우



pivot을 기준으로 분할된
두 subproblem은 서로 disjoint하다.

분할정복법 vs. 동적계획법

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.

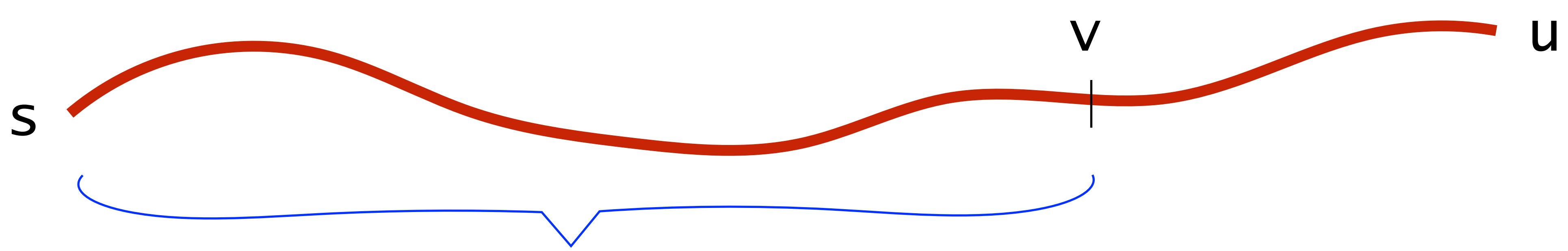
Optimal Substructure

- 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

이 경로가 s에서 u까지의 최단경로라면



이 경로는 s에서 v까지의 최단경로이다.

- 순환식은 optimal substructure를 표현한다.

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

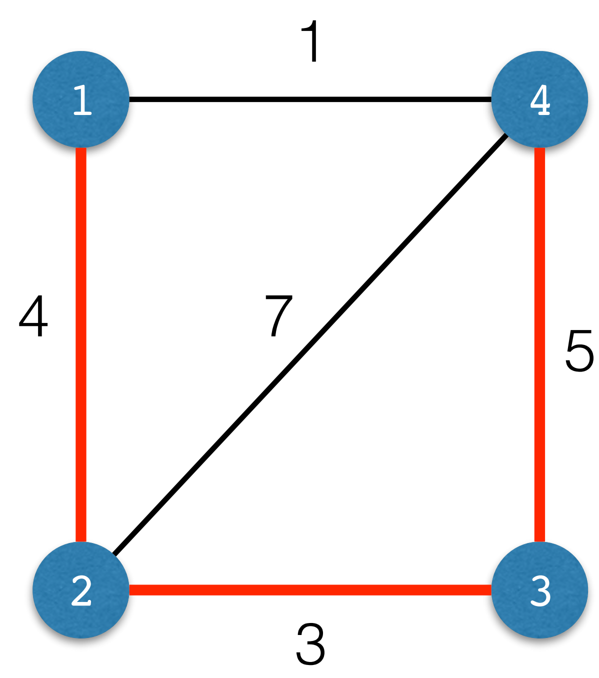
u까지 가는 최단경로의 길이는

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

Optimal Substructure를 확인하는 질문

- 최장경로(Longest-Path) 문제
  - 노드를 중복 방문하지 않고 가는 가장 긴 경로
  - optimal substructure를 가지는가?

최장경로문제



1에서 4까지의 최장경로는 (1,2,3,4)
하지만 1에서 3까지의 최장경로는 (1,4,2,3)

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

u까지 가는 최장경로가 v를 지난다고 하더라고 그 경로상에서 v까지 가는 경로가 반드시 v까지 가는 최장경로가 아닐수도 있으므로 이런 순환식은 성립하지 않는다.

그럼 최장경로 문제는 optimal substructure를 갖지 않는 것일까?

최장경로문제

s에서 집합 A에 속한 어떤 노드도 지나지 않고 u까지 가는 경로들 중에서 최장 경로의 길이

$$d(v, A) = \begin{cases} -\infty & \text{if } v \in A; \\ 0 & \text{if } v = s; \\ \max_{u \text{ adjacent to } v} \{d(u, A \cup \{v\}) + w(u, v)\} & \text{otherwise.} \end{cases}$$

즉 최장경로 문제는 다른 형태의 optimal substructure를 가지는 것일 뿐 optimal substructure를 가지지 않는다고 말할 수는 없다.?

Matrix-Chain Multiplication

- p×q 행렬 A와 q×r 행렬 B 곱하기

```
void product(int A[][], int B[][], int C[][]) {
    for (int i=0; i<p; i++) {
        for (int j=0; j<r; j++) {
            C[i][j] = 0;
            for (int k=0; k<q; k++)
                C[i][j] += A[i][k]*B[k][j;]
        }
    }
}
```

곱셈연산의 횟수 = pqr

Matrix-Chain 곱하기

- 행렬 A는 10×100, B는 100×5, C는 5×50
- 세 행렬의 곱 ABC는 두 가지 방법으로 계산가능 (결합법칙이 성립)
  - (AB)C : 7,500번의 곱셈이 필요 (10×100×5 + 10×5×50)
  - A(BC) : 75,000번의 곱셈이 필요 (100×5×50 + 10×100×50)
- 즉 곱하는 순서에 따라서 연산량이 다름
- n개의 행렬의 곱 $A_1A_2A_3\cdots A_n$을 계산하는 최적의 순서는?
- 여기서 A_i는 $p_{k-1} \times p_k$ 행렬이다.

Optimal Substructure

$$\underbrace{\underbrace{A_1 \; A_2 \; \cdots \; A_k}_{X} \;\; \underbrace{A_{k+1} \; A_{k+2} \; \cdots \; A_n}_{Y}}_{Z}$$

X는 앞부분의 곱이고
Y는 뒷부분의 곱이다.

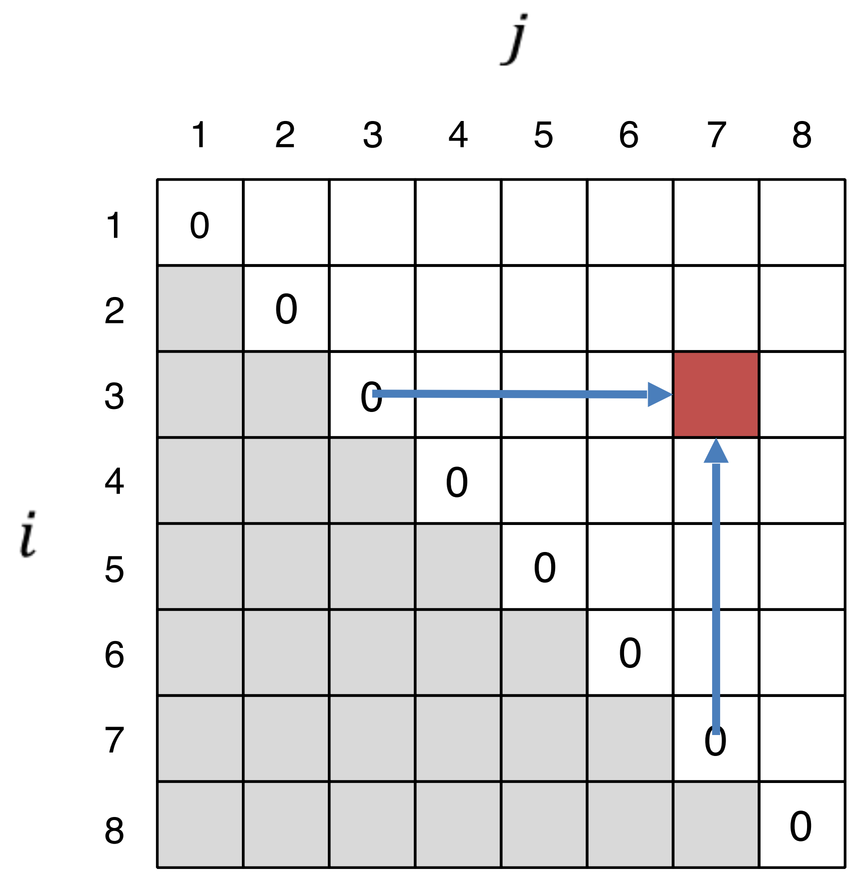
최종 결과 Z는 직전의 두 행렬 X와 Y의 곱이다.

순환식

- $m[i,j]$: $A_iA_{i+1}\cdots A_j$ 를 곱하는 최소곱셈 횟수

$$m[i,j] = \begin{cases} 0 & \text{if } i = j; \\ \min_{i \le k \le j-1}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & \text{if } i < j. \end{cases}$$

계산 순서



$$
m[i,j] = \begin{cases} 0 & \text{if } i = j; \\ \min_{i \le k \le j-1} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & \text{if } i < j. \end{cases}
$$

동적계획법

```
int matrixChain(int n)
{
    for (int i=1; i<=n; i++)
        m[i][i] = 0;
    for (int r=1; r<=n-1; r++) {
        for (int i = 1; i <= n - r; i++) {
            int j = i + r;
            m[i][j] = m[i+1][j] + p[i-1]*p[i]*p[j];
            for (int k = i+1; k <= j-1; k++) {
                if (m[i][j] > m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j])
                    m[i][j] = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            }
        }
    }
    return m[1][n];
}
```

시간복잡도: $\Theta(n^3)$

Longest Common Subsequence

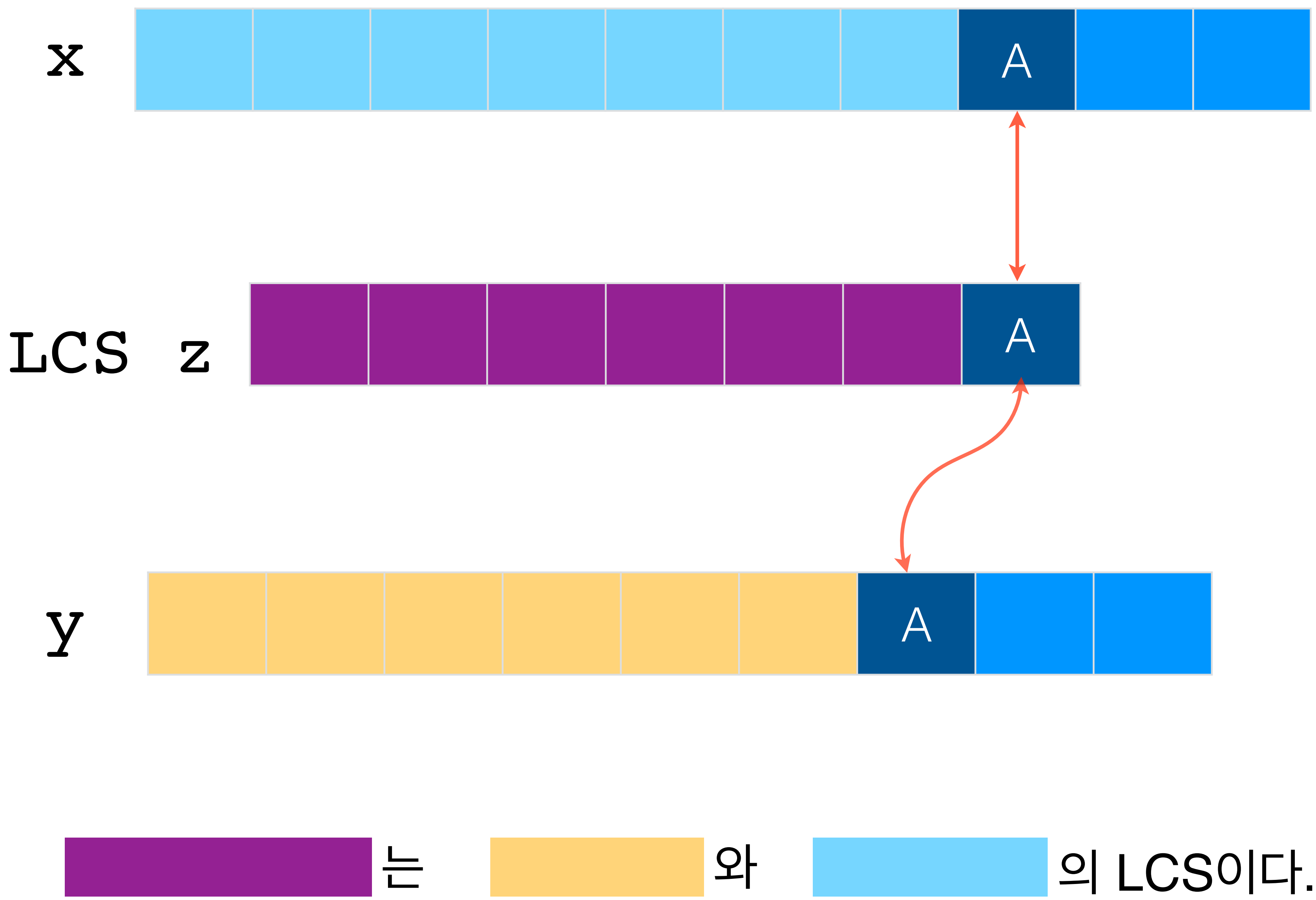
Longest Common Subsequence(LCS)

- <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

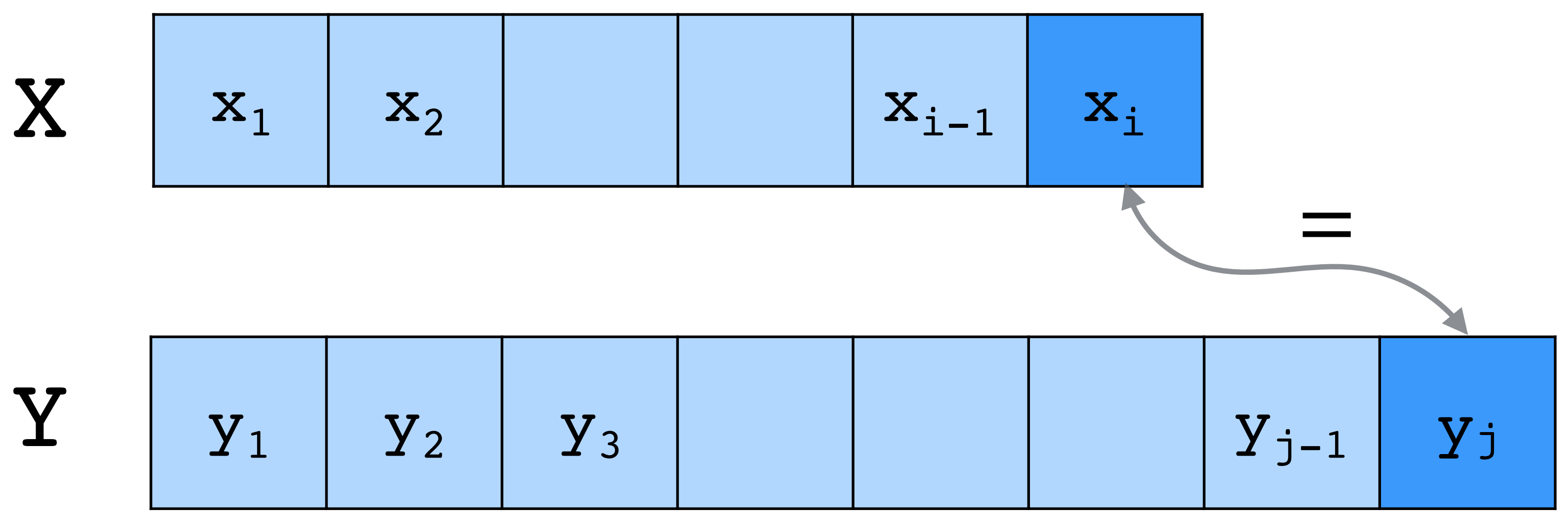
- 문자열 x의 모든 subsequence에 대해서 그것이 y의 subsequence가 되는지 검사한다.
- $|x|=m$, $|y|=n$
- x의 subsequence의 개수 = 2^m
- 각각이 y의 subsequence인지 검사: $O(n)$시간
- 시간복잡도 $O(n2^m)$

Optimal Substructure



순환식

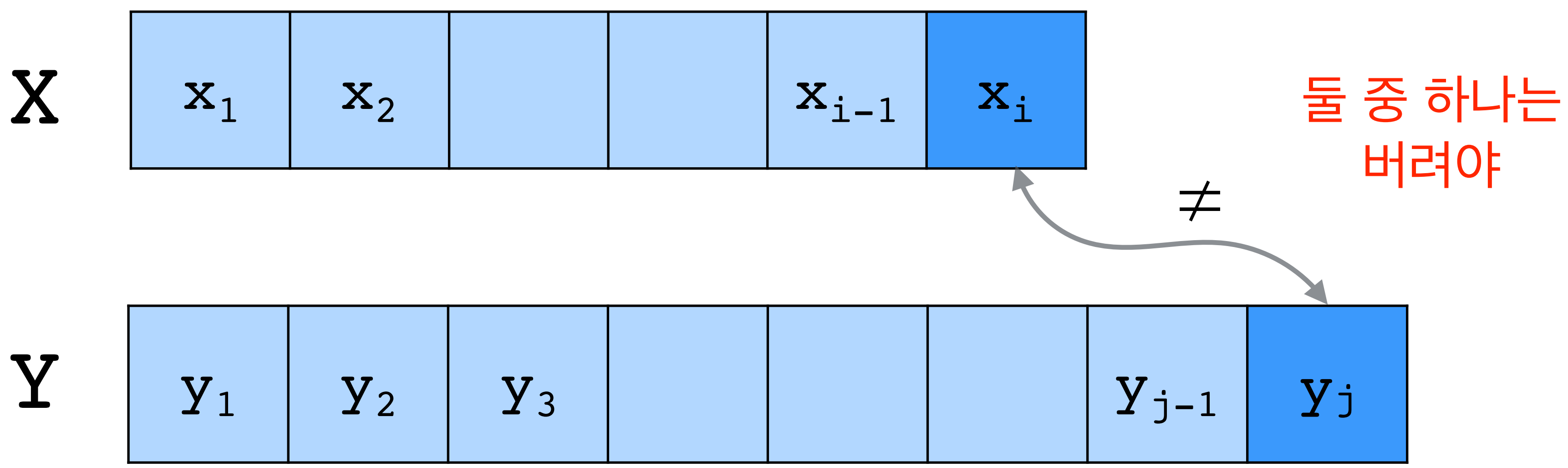
- $L[i,j]$: 문자열 $X =< x_1 x_2 \cdots x_i >$와 $Y =< y_1 y_2 \cdots y_j >$의 LCS의 길이



- 경우 1: $x_i = y_j$

$$L[i,j] = L[i-1, j-1] + 1$$

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



- 경우 2: $x_i \neq y_j$

$$L[i,j] = \max(L[i-1,j], L[i,j-1])$$

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

	j	0	1	2	3	4	5	6
i		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

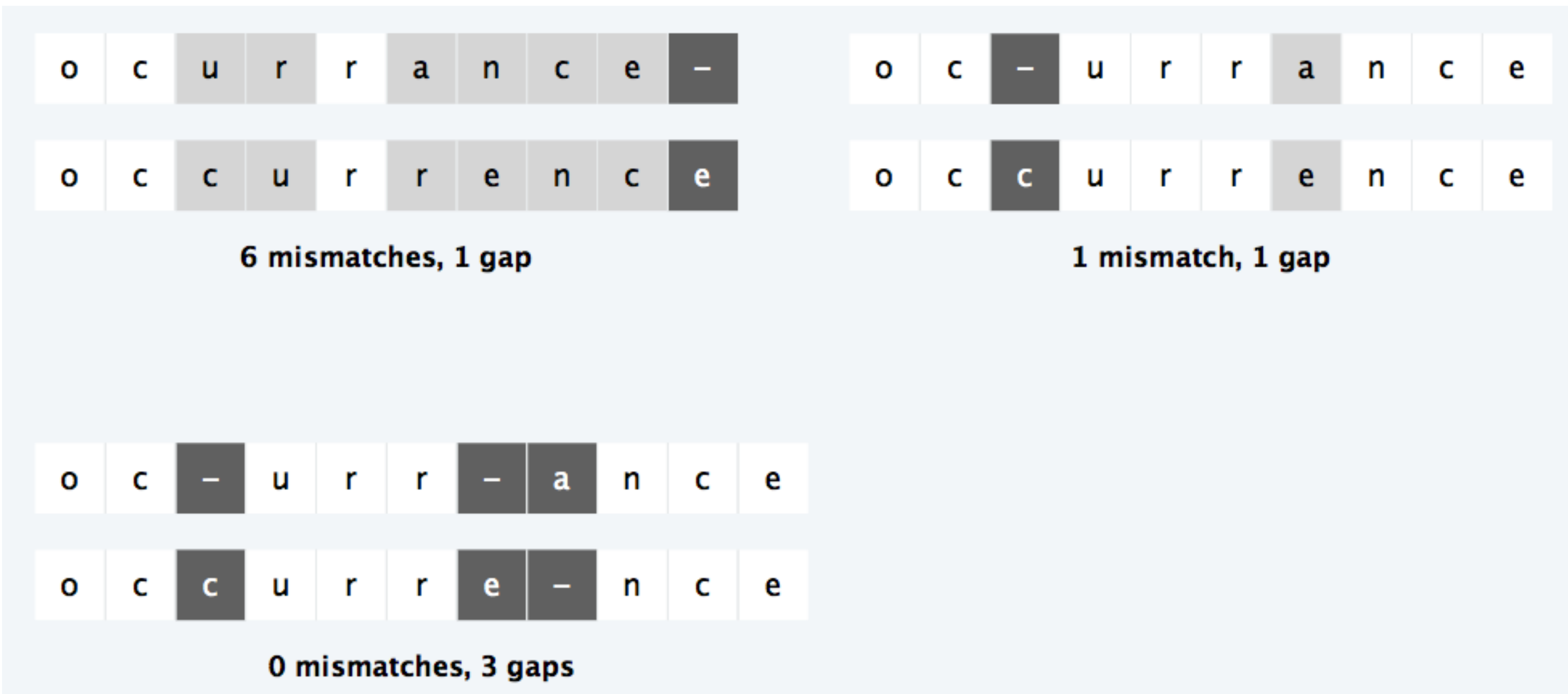
X = ABCBDAB
Y = BDCABA

```
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: $\Theta(mn)$

Shortest Edit Distance

- 두 문자열이 얼마나 유사한가 혹은 다른가?
- 예: “ocurrance”와 “occurrence”



Edit Distance

- 언매치(unmatch)
  - 어떤 문자가 자신과 대응되는 문자가 없는 경우
  - unmatch penalty δ;
- 미스매치(mismatch)
  - 어떤 문자 p가 자신과 다른 문자 q와 대응되는 경우
  - mismatch penalty α_{pq}
- Edit Distance = unmatch와 mismatch penalty의 총합

C	T	–	G	A	C	C	T	A	C	G
C	T	G	G	A	C	G	A	A	C	G

$$\text{cost} = \delta + \alpha_{CG} + \alpha_{TA}$$

Shortest Edit Distance

- 두 문자열 $x_1x_2...x_m$과 $y_1y_2...y_n$
- 두 문자열의 edit distance는 문자들을 어떻게 대응시키느냐에 따라 다름
- Edit distance의 최소값을 구하라.

Optimal substructure

- OPT(i,j) : 스트링 $x_1x_2...x_i$와 $y_1y_2...y_j$의 SED
- 3가지 경우:
  - 경우 1: x_i와 y_j를 대응시킬 경우
  - 경우 2: x_i를 아무와도 대응시키지 않을 경우
  - 경우 2: y_j를 아무와도 대응시키지 않을 경우

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i=0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j=0 \end{cases}$$

SEQUENCE-ALIGNMENT $(m, n, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha)$

FOR $i = 0$ TO m

$\quad M[i, 0] \leftarrow i\delta.$

FOR $j = 0$ TO n

$\quad M[0, j] \leftarrow j\delta.$

FOR $i = 1$ TO m

$\quad$ FOR $j = 1$ TO n

$$M[i, j] \leftarrow \min \{ \alpha[x_i, y_j] + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1]).$$

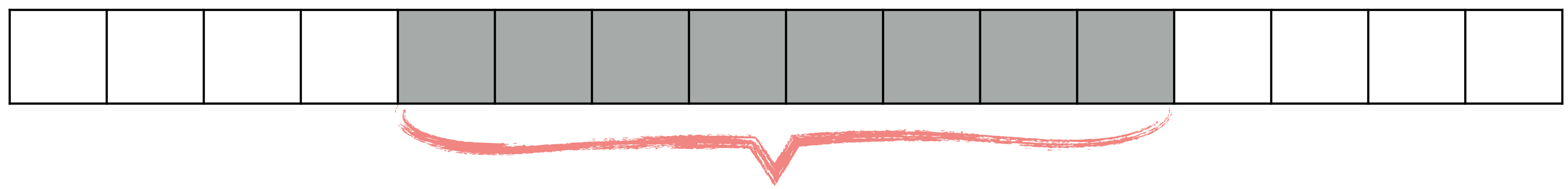
RETURN $M[m, n]$.

Maximum Sum Interval

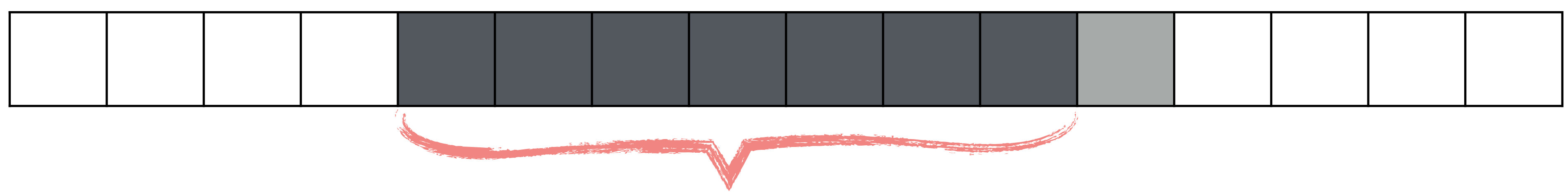
Maximum Sum Interval

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

Optimal Substructure



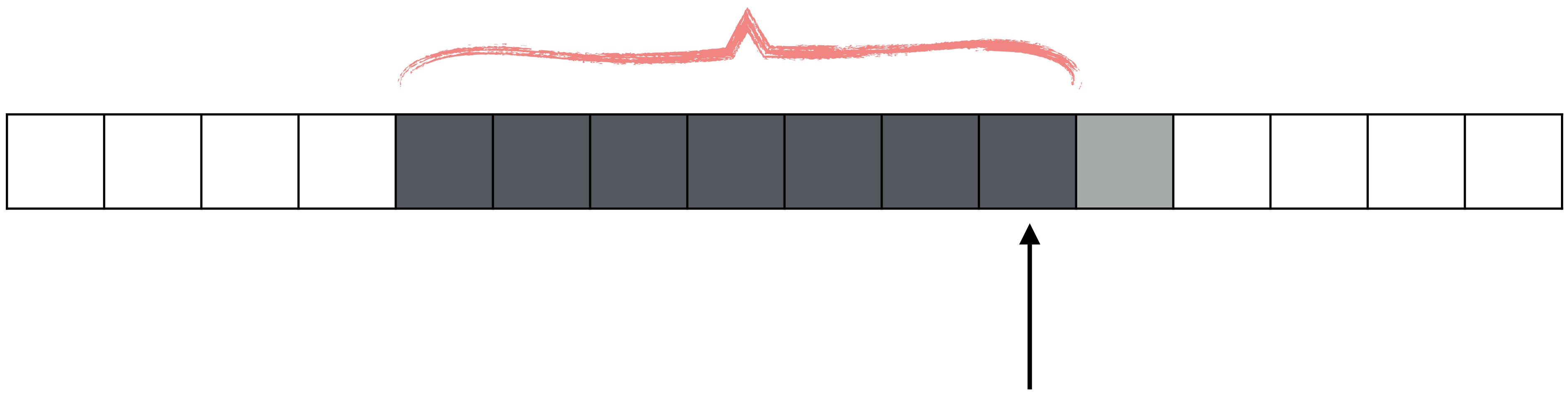
가령 이것이 합이 최대가 되는 구간이라고 가정해보자.



그렇다면 최적 구간의 일부인 이 구간의 정체는?

Optimal Substructure

그렇다면 최적 구간의 일부인 이 구간의 정체는?



끝점이 여기인 구간들 중에 합이 최대인 구간이다.

순환식

- $\text{maxEndsAt}[i]$: 끝점이 i인 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 = $\max_{i=1,2,\ldots,n} \text{maxEndsAt}[i]$

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 = Math.max(maxEndsAt[i-1] + A[i], A[i]);
        if (maxEndsAt > max)
            max = maxEndsAt;
    }
    return max;
}
```

시간복잡도 O(n)

Knapsack Problem

Knapsack

- n개의 아이템과 배낭
- 각각의 아이템은 무게 w_i와 가격 v_i를 가짐
- 배낭의 용량 W
- 목적: 배낭의 용량을 초과하지 않으면서 가격이 최대가 되는 부분집합
- 예:
  - {1,2,5}는 가격의 합이 35
  - {3,4}는 가격의 합이 40
  - {3,5}는 46이지만 배낭의 용량을 초과함

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

Greedy

- 가격이 높은 것 부터 선택
- 무게가 가벼운 것부터 선택
- 단위 무게당 가격이 높은것 부터 선택

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

순환식

- OPT(i) : 아이템 1,2,...,i로 얻을 수 있는 최대 이득
- 경우 1: 아이템 i를 선택하지 않는 경우
  - OPT(i) = OPT(i-1)
- 경우 2: 아이템 i를 선택하는 경우
  - OPT(i) = ?

- OPT(i, w) : 배낭 용량이 w일 때 아이템 1,2,...,i로 얻을 수 있는 최대 이득
- 경우 1: 아이템 i를 선택하지 않는 경우
  - OPT(i, w) = OPT(i-1, w)
- 경우 2: 아이템 i를 선택하는 경우
  - OPT(i) = v_i + OPT(i-1, w-w_i)

$$OPT(i,w) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ OPT(i-1,w) & \text{if } \mathrm{w_i} > \mathrm{w} \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \,\} & \text{otherwise} \end{cases}$$

Bottom-Up

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}.$

RETURN $M[n, W]$.

예

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

시간복잡도

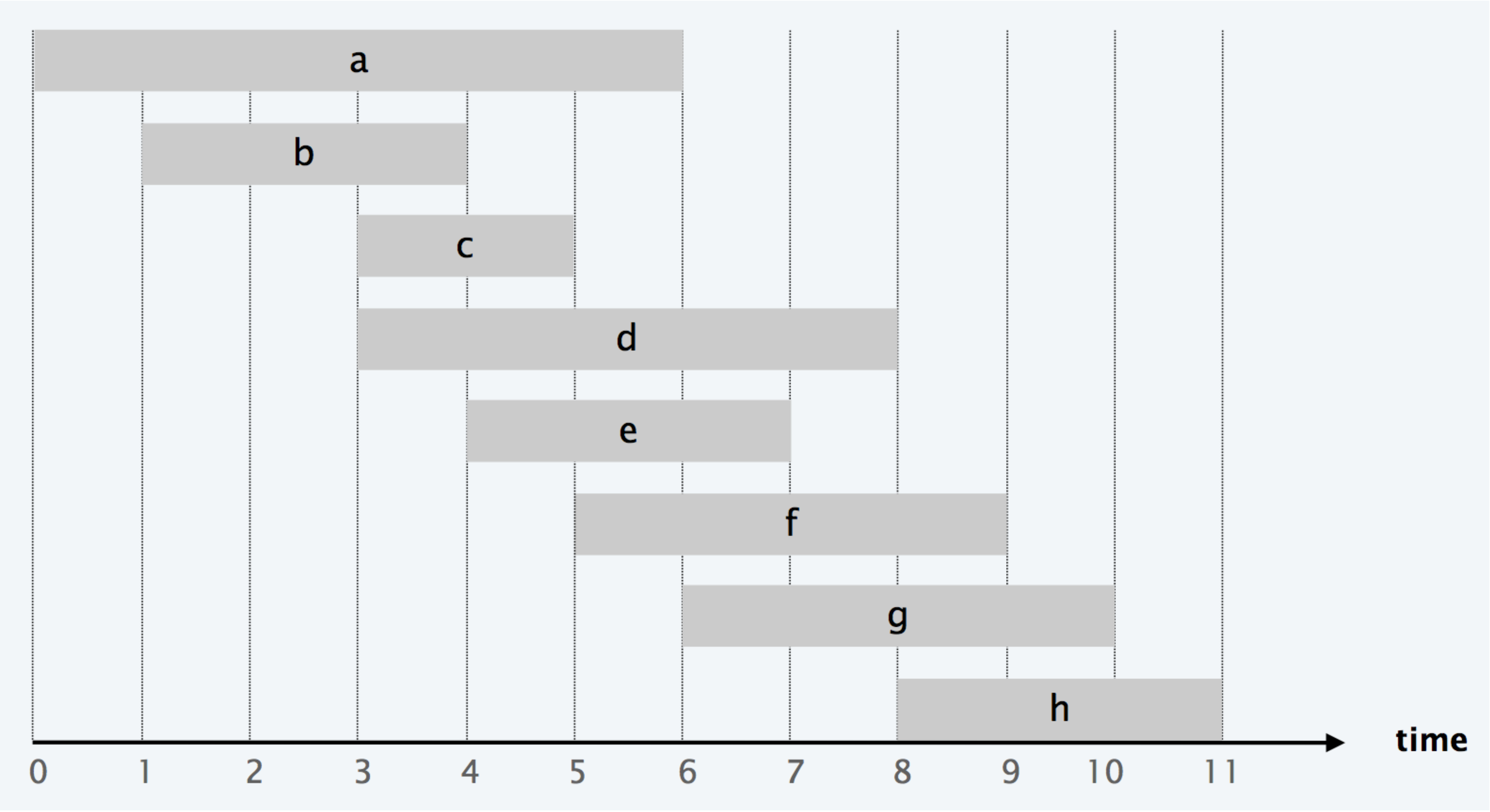
- 시간복잡도 O(nW)
- 다항시간인가?

Weighted Interval Scheduling

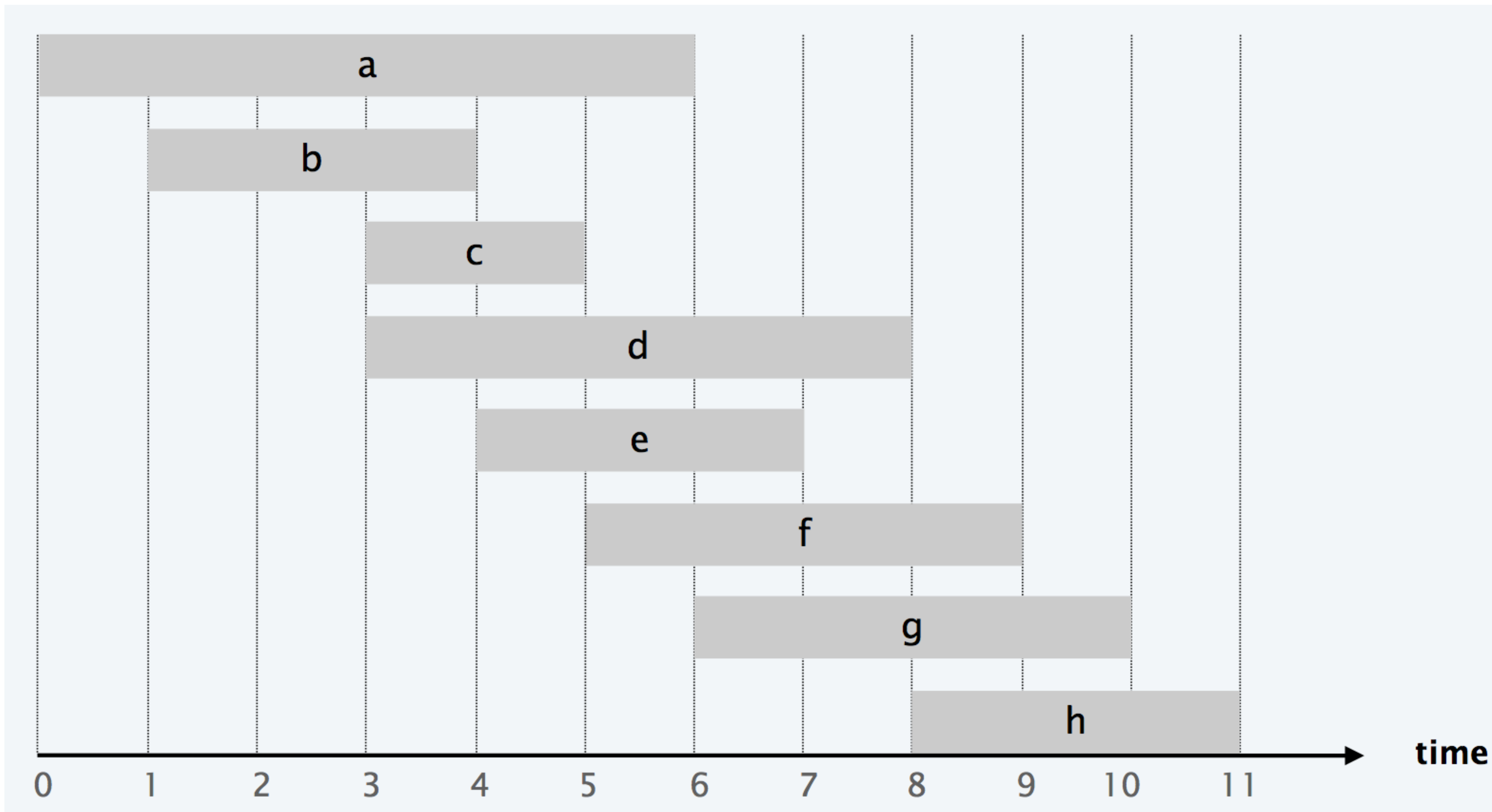
Weighted Interval Scheduling

- N개의 작업이 주어짐. 각각의 작업(job)은 시작 시각, 종료 시각, 그리고 가중치를 가짐
- 즉 작업 j는 (s_j, f_j, w_j)로 표현됨, 여기서 s_j는 시작 시각, f_j는 종료시각, 그리고 w_j는 가중치
- 시간적으로 겹치지 않는 두 작업은 서로 compatible하다고 말함
- 서로 compatible하면서 가중치의 합이 최대가 되는 부분집합을 찾아라.

Weighted Interval Scheduling

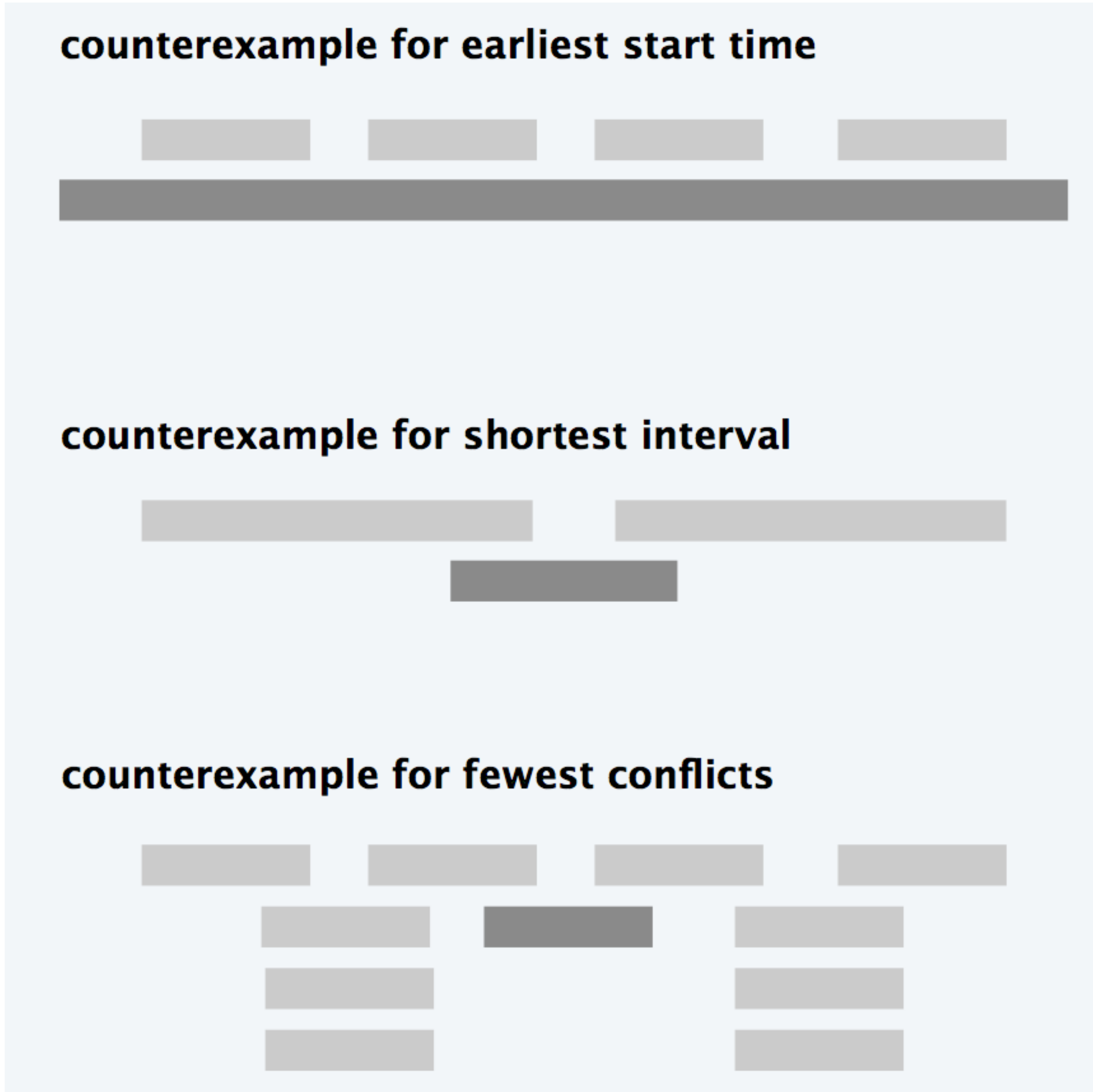


모든 작업의 가중치가 1이라면?



서로 compatible한 최대개수의 부분집합을 찾는 문제

모든 작업의 가중치가 1이라면?

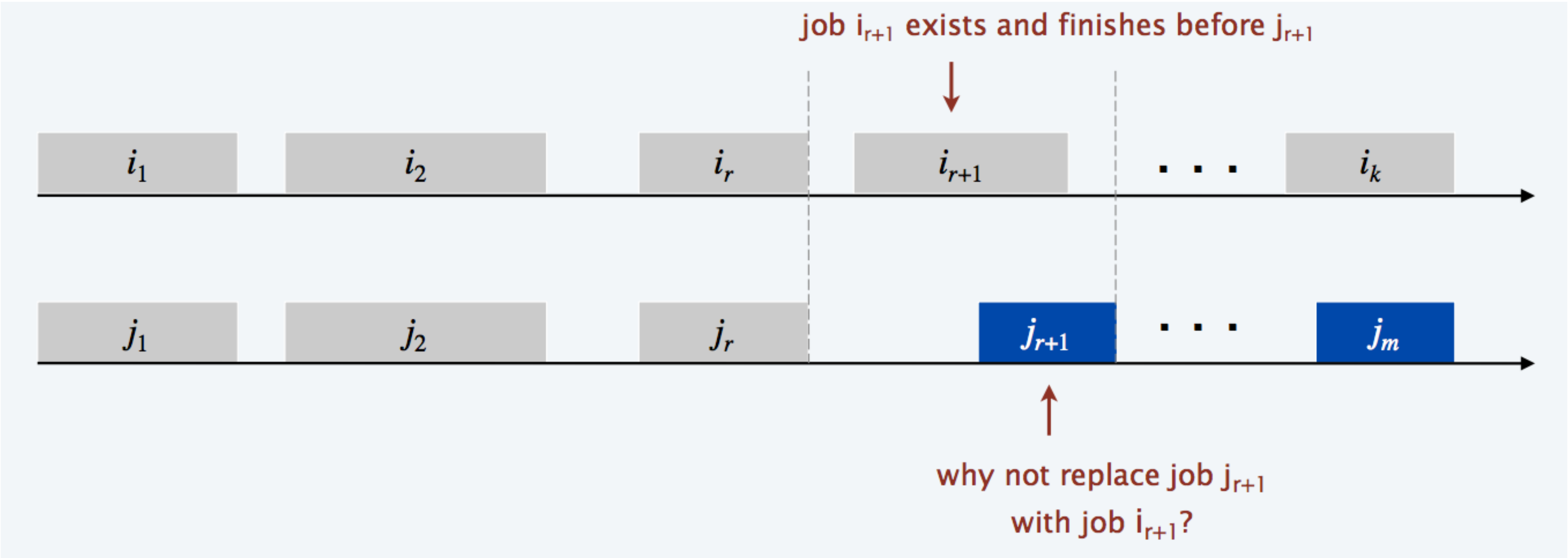


Earliest-Finish-Time First (EFTF)

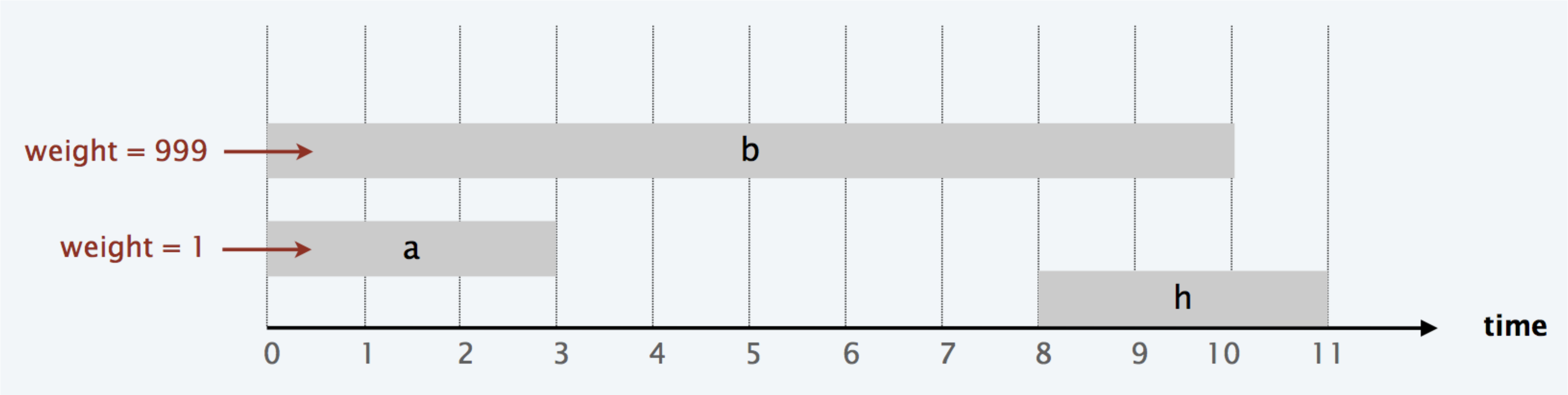
- Finish Time이 빠른 것 부터 순서대로 고려한다.
- 이미 선택한 작업들과 compatible하면 선택한다.

EFTF의 최적성 증명

- 최적이 아니라고 가정하자.
- $i_1, i_2, \ldots, i_k$를 EFTF 알고리즘이 선택한 작업이라고 하고, $j_1, j_2, \ldots, j_m$을 최적해라고 하자.
- $i_1=j_1, i_2=j_2, \ldots, i_r=j_r$이고 $i_{r+1} \neq j_{r+1}$인 최대 인덱스를 r이라고 하자.

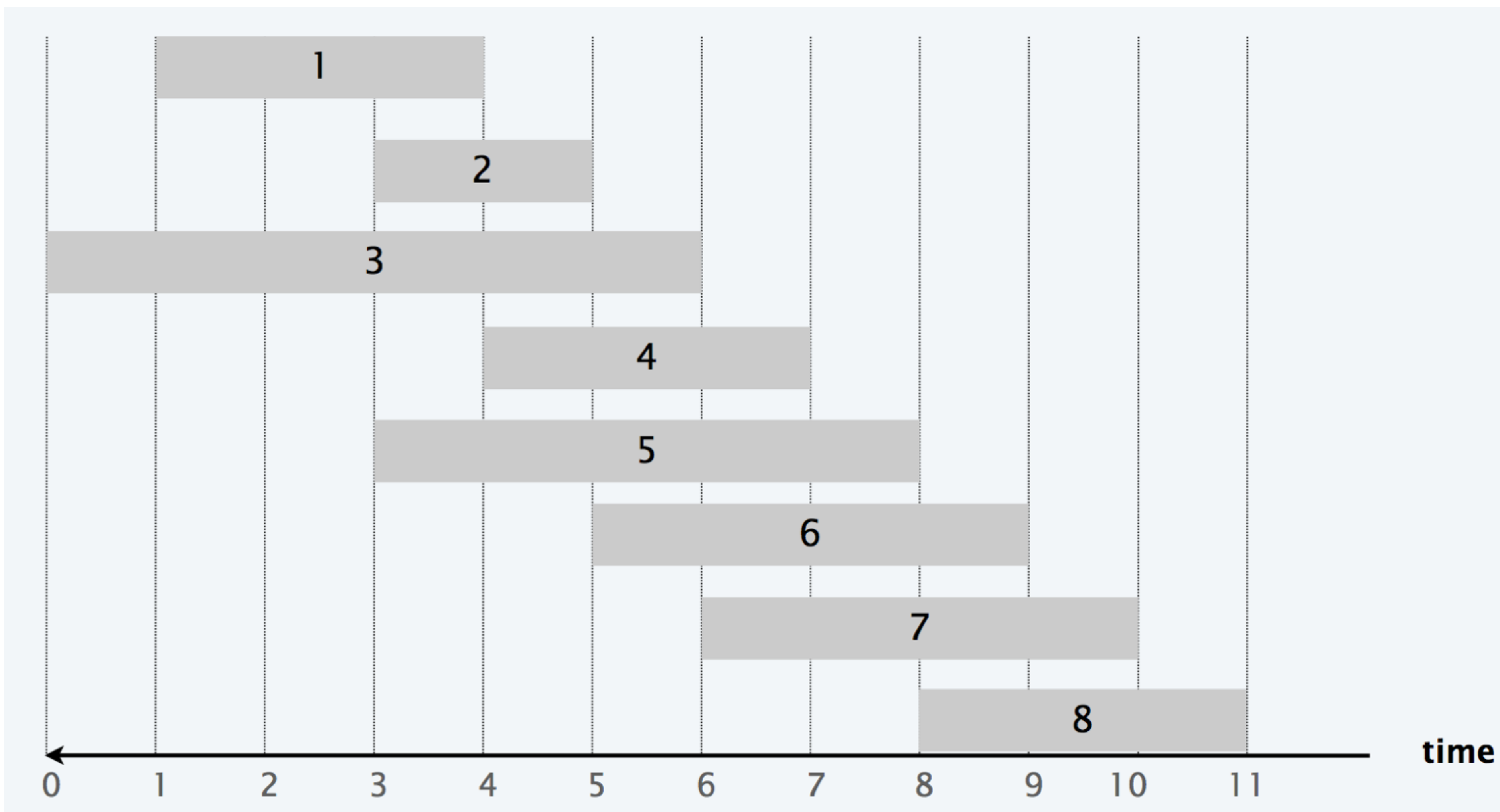


- 가중치가 있는 경우에는 성립하지 않음



가중치가 있는 경우

- 작업들이 종료시각을 기준으로 정렬되어 있다고 가정. 즉 f1<=f2<=...<=fn.
- p(j) = 작업 j와 compatible하면서 가장 늦은 종료시간을 가진 작업의 인덱스
- 예) p(8)=5, p(7)=3, p(2)=0.



순환식

- OPT(j): 작업들 1,2,...,j에 대한 최적해
- 경우 1: 작업 j를 선택하는 경우
  - 작업 j의 이득 v_j를 챙김
  - 작업들 {p(j)+1, p(j)+2,...,j-1}는 선택할 수 없음
  - 작업들 1,2,...,p(j)들 중에서 최적으로 선택함
- 경우 2: 작업 j를 선택하지 않는 경우
  - 작업들 1,2,...,j-1들 중에서 최적으로 선택함

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0.$

FOR j = 1 TO n

  $M[j] \leftarrow \max \{ v_j + M[p(j)],\ M[j-1] \}.$

How to compute p(i) ?

O(nlogn)