# Chapter 5 동적 계획 알고리즘

#### 차례

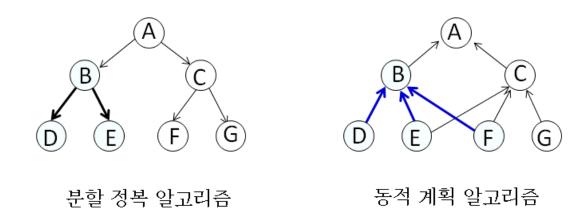
- 5.1 모든 쌍 최단 경로
- 5.2 연속 행렬 곱셈
- 5.3 편집 거리 문제
- 5.4 배낭 문제
- 5.5 동전 거스름돈

### 동적 계획 알고리즘

- ➤ Dynamic Programming(DP) 알고리즘
  - 입력 크기가 작은 부분 문제들을 해결한 후에
  - 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여
  - 최종적으로 원래 주어진 입력의 문제를 해결

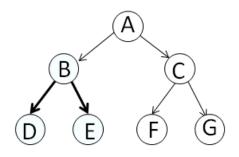
#### DP vs 분할 정복 알고리즘

▶ 분할 정복 알고리즘과 DP 알고리즘의 전형적인 부분 문제들 사이의 관계

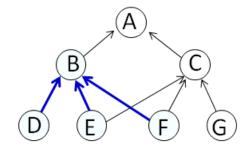


- \_ 분할 정복 알고리즘
  - A는 B와 C로 분할되고, B는 D와 E로 분할되는데, D와 E의 해를 취합하여 B의 해를 구한다.
    - 단, D, E, F, G는 각각 더 이상 분할할 수 없는 (또는 가장 작은 크기의) 부분 문제들이다.
  - 마찬가지로 F와 G의 해를 취합하여 C의 해를 구하고, 마지막으로 B와 C의 해를 취합하여 A의 해를 구한다.

#### DP vs 분할 정복 알고리즘



분할 정복 알고리즘



동적 계획 알고리즘

#### - DP 알고리즘

- 먼저 최소 단위의 부분 문제 D, E, F, G의 해를 각각 구한다.
- 그 다음 D, E, F의 해를 이용하여 B의 해를 구한다.
- E, F, G의 해를 이용하여 C의 해를 구한다.
- B와 C의 해를 계산하는데 E와 F의 해 모두를 이용한다.

#### DP 알고리즘

- ➤ DP 알고리즘에는 부분 문제들 사이에 의존적 관계가 존재
  - 작은 부분 문제의 해가 보다 큰 부분 문제를 해결하는데 사용되는 관계가 있다.
  - 이러한 관계는 문제 또는 입력에 따라 다르고,
     대부분의 경우 뚜렷이 보이지 않아서 '함축적 순서'
     (implicit order)라고 함
- ▶ 분할 정복 알고리즘은 부분 문제의 해를 중복 사용하지 않음

#### 5.1 모든 쌍 최단 경로 문제

- > 모든 쌍 최단 경로 (All Pairs Shortest Paths) 문제
  - 각 쌍의 점 사이의 최단 경로를 찾는 문제

	서 울 Seoul	인 천 Incheon	수 원 Suwon	대 전 Daejeon	전 주 Jeonju	광주 Gwangju	대구 Daegu	을 산 Ulsan	부 산 Busan
서 을 Seoul		40.2	41.3	154	232.1	320.4	297	407.5	432
인 천 Incheon			54.5	174	253.3	351.6	317.6	447	453
수 원 Suwon				132.6	189.4	299.6	268.1	356	390.7
대 전 Daejeon					96.9	185,2	148.7	259.1	283.4
전 주 Jeonju						105.9	219.7	331.1	322.9
광 주 Gwangju			1245000000000000000000000000000000000000				219.3	329.9	268
돼.구 Daegu								111.1	135.5
을 산 Ulsan									52.9
부 산 Busan									

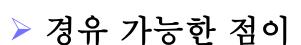
- > 다익스트라의 최단 경로 알고리즘 이용하면
  - 각 점을 시작점으로 정하여 다익스트라 알고리즘 수행
  - 시간 복잡도는 n x O(n²) = O(n³), 단, n은 점의 수

#### 모든 쌍 최단 경로 문제

- > 플로이드-워샬 알고리즘
  - 간단히 플로이드 알고리즘으로 부르자.
  - 플로이드 알고리즘의 시간 복잡도는 O(n³)으로 다익스트라 알고리즘을 n번 사용할 때의 시간 복잡도와 동일
  - 플로이드 알고리즘은 매우 간단하여 다익스트라 알고리즘보다 효율적



- > 작은 그래프에서 부분 문제들을 찾아보자.
  - 3개의 점이 있는 경우, a에서 c까지의 최단 경로를 찾으려면 2가지 경로, 즉, a에서 c로 직접 가는 경로와 점 b를 경유하는 경로 중에서 짠으 거은 서태



- 점 1인 경우
- 점 1, 2인 경우,
- 점 1, 2, 3인 경우

• • •

- 점 1, 2, ···, n, 즉 모든 점을 경유 가능한 점들로 고려하면서, 모든 쌍의 최단 경로의 거리를 계산

#### 부분 문제의 정의

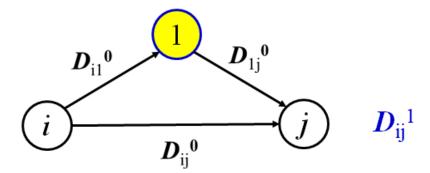
▶ 그래프의 점이 1, 2, 3, ···, n일 때

 $D_{ij}^{k}$  = 점  $\{1, 2, \dots, k\}$ 를 경유 가능한 점으로 고려하여, 점 i에서 점 j까지의 모든 경로 중에서 가장 짧은 경로의 거리

- [주의] 점 1에서 점 k까지의 <u>모든 점을 반드시</u> 경유하는 경로를 의미하는 것이 아니다.
- D<sub>ij</sub>k는 {1, 2, ···, k}을 하나도 경유하지 않으면서 점 i
   에서 직접 점 j에 도달하는 간선 (i, j)가 가장 짧은 거리일수도
- 단, k≠i, k≠j이고
- k=0인 경우, 점 0은 그래프에 없으므로 어떤 점도 경유하지 않는다는 것을 의미. 즉,  $D_{ij}^{0}$  = 간선 (i,j)의 가중치

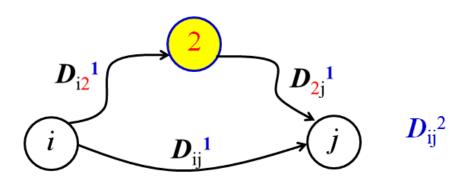
### $D_{ij}^{-1}$

- 점 i에서 점 j까지 점 1을 경유하는 경우와 직접 가는 경로 중에서 짧은 경로의 거리
- ightharpoonup 모든 점 i와 j에 대해  $D_{ij}$ 1를 계산하는 것이 가장 작은 부분 문제
- $\rightarrow$  i  $\neq$  1, j  $\neq$  1



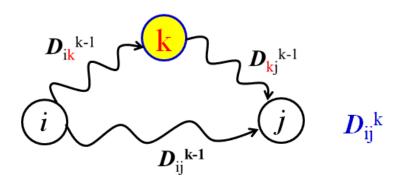
$$D_{ij}^2$$

- ightharpoonup 점 i에서 점 2를 경유하여 j로 가는 경로의 거리와  $D_{ij}^{1}$  중에서 짧은 경로의 거리
- ightharpoonup 단, 점 2를 경유하는 경로의 거리는  $D_{i2}^{1} + D_{2i}^{1}$
- $\geq$  i  $\neq$  2, j  $\neq$  2



### $D_{ij}^{k}$

- ightharpoonup 점 i에서 점 k를 경유하여 j로 가는 경로의 거리와  $D_{ij}^{k-1}$  중에서 짧은 경로의 거리
- ightharpoonup 점 k를 경유하는 경로의 거리는  $D_{ik}^{k-1} + D_{kj}^{k-1}$  이고,  $i \neq k$ ,  $j \neq k$



### $D_{ij}$ n

▶ 모든 점을 경유 가능한 점들로 고려한 모든 쌍 i와 j의 최단 경로의 거리

ightharpoonup 플로이드의 모든 쌍 최단 경로 알고리즘은 k가 1에서 n이 될 때까지  $D_{ii}$  를 계산해서,  $D_{ii}$  을 찾는다.

# 알고리즘

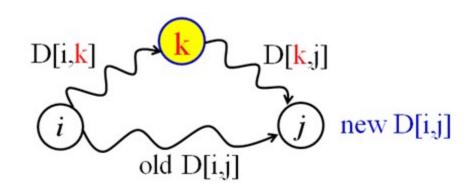
#### **AllPairsShortest**

입력: 2차원 배열 D, 단, D[i, j] = 간선 (i, j)의 가중치, 만일 간선 (i, j)가 없으면 D[i, j] = ∞, 모든 i에 대하여 D[i, i] = 0 출력: 모든 쌍 최단 경로의 거리를 저장한 2차원 배열 D



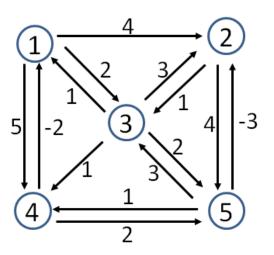
- 1. for k = 1 to n
- 2. for i = 1 to  $n (i \neq k)$
- 3. for j = 1 to  $n (j \neq k, j \neq i)$
- 4.  $D[i, j] = min\{D[i, k] + D[k, j], D[i, j]\}$

#### 부분 문제 간의 함축적 순서



▶ D[i, j]를 계산하기 위해서 미리 계산되어 있어야 할 부분 문제는 D[i, k]와 D[k, j]이다.

### 수행 과정



D	1	2	3	4	5
1	0	4	2	5	8
2	8	0	1	8	4
3	1	3	0	1	2
4	-2	∞	∞	0	2
5	∞	-3	3	1	0

#### k = 1일 때

- 
$$D[2,3] = min\{D[2,3], D[2,1] + D[1,3]\} = min\{1, \infty + 2\} = 1$$

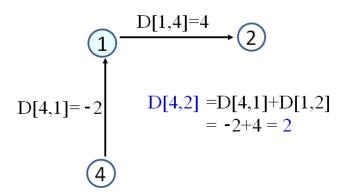
- 
$$D[2,4] = min\{D[2,4], D[2,1] + D[1,4]\} = min\{\infty, \infty + 5\} = \infty$$

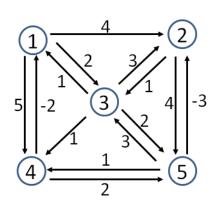
- 
$$D[2,5] = min\{D[2,5], D[2,1] + D[1,5]\} = min\{4, \infty + \infty\} = 4$$

$$-D[3,2] = min\{D[3,2], D[3,1] + D[1,2]\} = min\{3, 1+4\} = 3$$

$$-D[3,4] = min\{D[3,4], D[3,1] + D[1,4]\} = min\{1, 1+5\} = 1$$

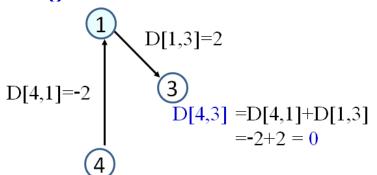
- 
$$D[3,5] = min\{D[3,5], D[3,1] + D[1,5]\} = min\{2, 1 + \infty\} = 2$$

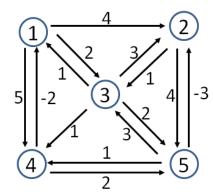




#### k = 1일 때

- D[4,3] = min{D[4,3], D[4,1]+D[1,3]} = min{∞, -2+/2}갱신됨 = 0





- D[4,5] =  $\min\{D[4,5], D[4,1]+D[1,5]\} = \min\{2, -2+\infty\}$ = 2
- D[5,2] = min{D[5,2], D[5,1]+D[1,2]} = min{-3, ∞+4} = -3
- D[5,3] = min{D[5,3], D[5,1]+D[1,3]} = min{3,  $\infty$ + 2} = 3

 $D[5,1] = min\{D[5,1], D[5,1] + D[1,1]\} = min\{1,\infty+5\} =$ 

#### k = 1일 때 수행 결과

- k=1일 때 D[4, 2], D[4, 3]이 각각 2, 0으로 갱신, 다른 원소들은 변하지 않음

D	1	2	3	4	5
1	0	4	2	5	8
2	8	0	1	8	4
3	1	3	0	1	2
4	-2	8	∞	0	2
5	$\infty$	-3	3	1	0



D	1	2	3	4	5
1	0	4	2	5	$\infty$
2	$\infty$	0	1	$\infty$	4
3	1	3	0	1	2
4	-2	2	0	0	2
5	$\infty$	-3	3	1	0

#### k = 2일 때

- D[1,5]가 1 → 2 → 5의 거리인 8로 갱신
- D[5,3]이 5 → 2 → 3의 거리인 -2로 갱신

D	1	2	3	4	5
1	0	4	2	5	$\infty$
2	$\infty$	0	1	$\infty$	4
3	1	3	0	1	2
4	-2	2	0	0	2
5	$\infty$	-3	3	1	0



D	1	2	3	4	5
1	0	4	2	5	8
2	$\infty$	0	1	$\infty$	4
3	1	3	0	1	2
4	-2	2	0	0	2
5	$\infty$	-3	-2	1	0

### k = 3일 때

#### - 총 7개의 원소가 갱신

D	1	2	3	4	5
1	0	4	2	5	8
2	$\infty$	0	1	$\infty$	4
3	1	3	0	1	2
4	-2	2	0	0	2
5	$\infty$	-3	-2	1	0



D	1	2	3	4	5
1	0	4	2	3	4
2	2	0	1	2	3
3	1	3	0	1	2
4	-2	2	0	0	2
5	-1	-3	-2	-1	0

### k = 4일 때

#### - 총 3개의 원소가 갱신

D	1	2	3	4	5
1	0	4	2	3	4
2	2	0	1	2	3
3	1	3	0	1	2
4	-2	2	0	0	2
5	- 1	-3	-2	- 1	0



D	1	2	3	4	5
1	0	4	2	3	4
2	0	0	1	2	3
3	-1	3	0	1	2
4	-2	2	0	0	2
5	-3	-3	-2	-1	0

### k = 5일 때

총 3개의 원소가 갱신되고, 이것이 주어진 입력에 대한 최종해

D	1	2	3	4	5
1	0	4	2	3	4
2	0	0	1	2	3
3	-1	3	0	1	2
4	-2	2	0	0	2
5	-3	-3	-2	-1	0



D	1	2	3	4	5
1	0	1	2	3	4
2	0	0	1	2	3
3	-1	-1	0	1	2
4	-2	-1	0	0	2
5	-3	-3	-2	-1	0

#### 시간 복잡도

▶ 각 k에 대해서 모든 i, j 쌍에 대해 계산되므로, 총 n x n x n = n³회 계산이 이루어지고, 각 계산은 O(1) 시간 소요

▶ 시간 복잡도는 O(n³)

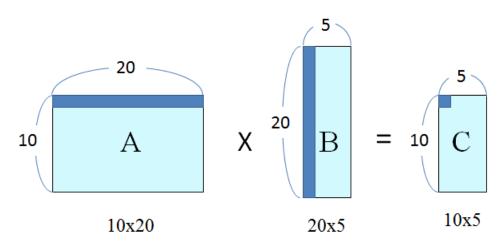


#### **Applications**

- ▶ 맵퀘스트(MapQuest)와 구글 맵
- > 자동차 네비게이션
- > 지리 정보 시스템 (GIS)에서의 네트워크 분석
- > 통신 네트워크와 모바일 통신 분야
- > 게임
- ▶ 산업 공학/경영 공학의 운영 (Operation) 연구
- > 로봇 공학
- > 교통 공학
- ▶ VLSI 디자인 분야 등

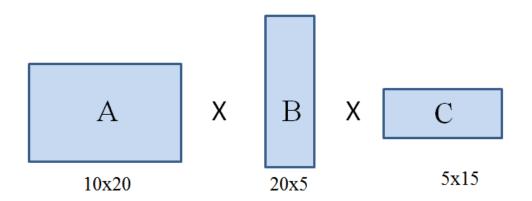
#### 5.2 연속 행렬 곱셈

- ▶ 연속 행렬 곱셈 (Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소 간의 최소 곱셈 횟수를 찾는 문제
- ➤ 10x20 행렬 A와 20x5 행렬 B를 곱하는데 원소 간의 곱셈 횟수는 10x20x5 = 1,000.
- > 두 행렬을 곱한 결과 행렬 C는 10x5



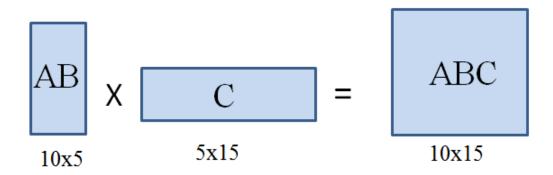
#### 행렬 곱셈

- > 3개의 행렬을 곱해야 하는 경우
- ▶ 연속된 행렬의 곱셈에는 결합 법칙 허용
- $\rightarrow$  AxBxC = (AxB)xC = Ax(BxC)



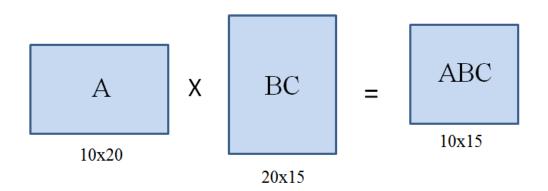
#### AXB를 계산한 후에 C를 곱하기

- ➤ AxB를 계산하는데 10x20x5 = 1,000번
- ▶ 결과 행렬의 크기가 10x5이고, 이에 행렬 C를 곱하면 10x5x15 = 750번
- ▶ 1,000 + 750 = 1,750회의 원소의 곱셈이 필요



#### BXC를 계산한 후에 A를 곱하기

- ➤ BxC를 계산하는데 20x5x15 = 1,500번
- ▶ 1,500 + 3,000 = 4,500회의 곱셈이 필요



- ▶ [주의] 주어진 행렬의 순서를 지켜서 반드시 이웃하는 행렬끼리 곱해야
  - AxBxCxDxE일 때
  - AxC를 수행하거나, AxD를 먼저 수행할 수 없음

## 부분 문제

부분 문제 크기						부분 문제 개수
1	A	В	$\mathbf{C}$	D	E	5개
2	AxB	Bx	C C>	kD I	DxE	4개
3	AxB	KC B	SXCXD	CxDx	Œ	3개
4	Ax	BxCxI	) Bx	CxDx1	E	2개
5	1	AxB	хСхІ	ЭxЕ		1개

# 알고리즘

```
입력: 연속된 행렬 A<sub>1</sub>xA<sub>2</sub>x···xA<sub>n</sub>,
      단, A_1은 d_0xd_1, A_2는 d_1xd_2, ..., A_n은 d_{n-1}xd_n이다.
출력: 입력의 행렬 곱셈에 필요한 원소의 최소 곱셈 횟수
1. for i = 1 to n
2. C[i, i] = 0
3. for L = 1 to n-1 // L은 부분 문제의 크기를 조절하는 인덱스이다.
4. for i = 1 to n-L
5. j = i + L
6.
          C[i, j] = \infty
7.
          for k = i to i-1
                temp = C[i, k] + C[k+1, j] + d_{i-1}d_kd_i
8.
9.
                if (temp < C[i, j])
                       C[i, j] = temp
10.
11. return C[1,n]
```

### Line 3의 for-루프

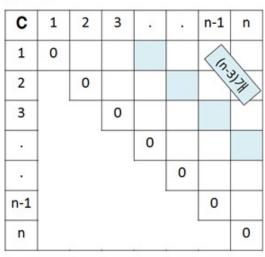
L = 1

С	1	2	3			n-1	n
1	0						
2		0			1		
3			0			W.	
				0		Ť	
					0		
n-1						0	
n							0

L = 2

С	1	2	3			n-1	n
1	0				- 15		
2		0				(n.2)	
3			0			THE STREET	>
				0			
					0		
n-1						0	
n							0

L = 3



L = n-2

С	1	2	3			n-1	n
1	0						
2		0			<	3	
3			0				
				0			
					0		
n-1						0	
n							0

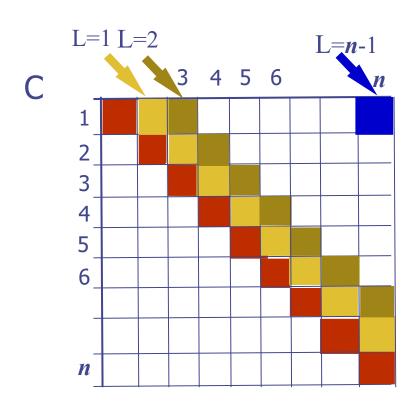
L = n-1

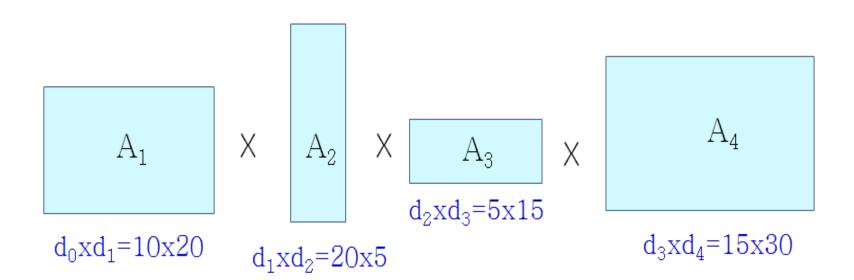
С	1	2	3	•	٠.	n-1	n
1	0						
2		0			_	13	-
3			0			~	
				0			
					0		
n-1						0	
n							0

• • •

#### Line 7의 for-루프

## 알고리즘 수행 순서





## L=1일 때

i = 2

i = 3

$$ightharpoonup C[1, 2] = d_0 d_1 d_2 = 10x20x5 = 1,000$$
  $i = 1$ 

$$ightharpoonup C[2, 3] = 20x5x15 = 1,500$$

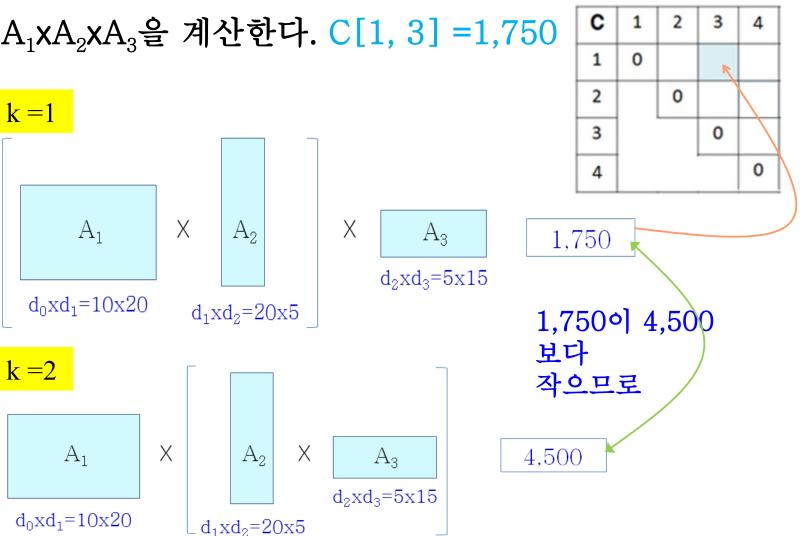
 $\triangleright$  C[3, 4] = 5x15x30 = 2,250

	4

С	1	2	3	4
1	0	4		
2		0	<b>L</b>	
3			0	
4				0

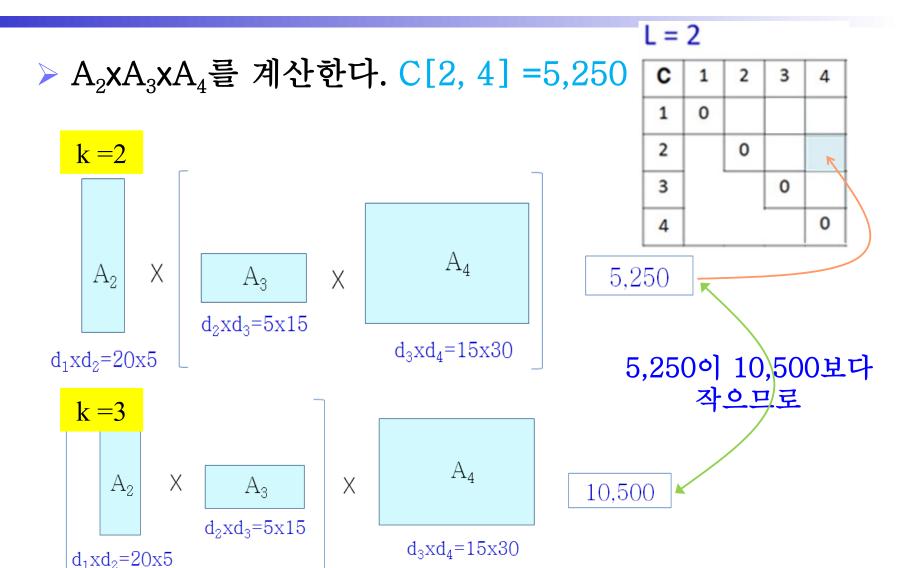
# L=2, i=1일 때

 $\rightarrow$  A<sub>1</sub>xA<sub>2</sub>xA<sub>3</sub>을 계산한다. C[1, 3] =1,750



L = 2

# L=2, i=2일 때



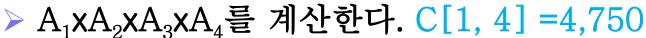
#### L=3일 때

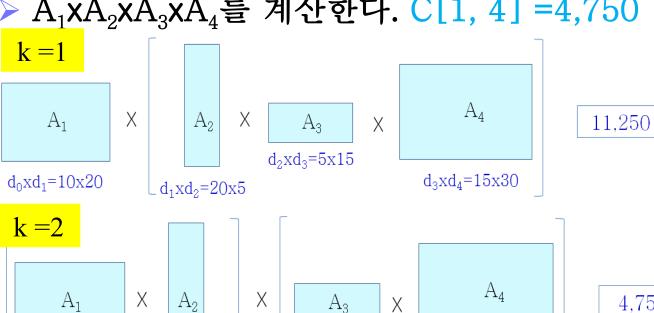
L = 3

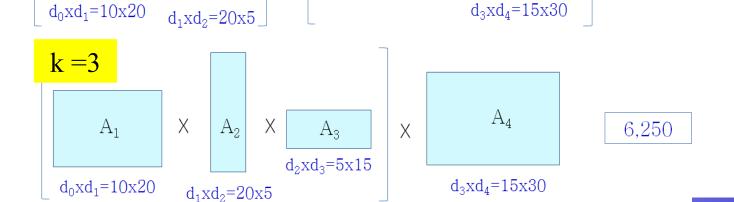
가장 작으므로

C

4,750







 $d_2xd_3=5x15$ 

# 수행 결과

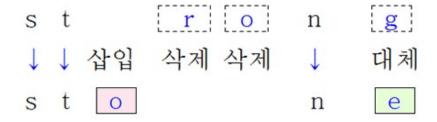
С	1	2	3	4
1	0	1,000	1,750	4,750
2		0	1,500	5,250
3			0	2,250
4				0

# 시간 복잡도

- ➤ 총 부분 문제 수: (n-1) + (n-2) + ··· + 2 + 1 = n(n-1)/2
- ▶ 하나의 부분 문제는 k-루프가 최대 (n-1)번 수행
- ▶ 시간 복잡도: O(n²) x O(n) = O(n³)

#### 5.3 편집 거리 문제

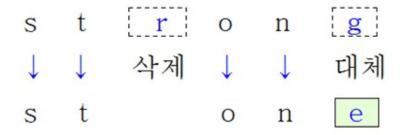
- ➤ 편집 거리 (Edit Distance): 삽입 (insert), 삭제 (delete), 대체 (substitute) 연산을 사용하여 스트링(문자열) S를 수정하여 다른 스트링 T로 변환하고자 할 때 필요한 최소의 편집 연산 횟수
- 'strong' ⇒ 'stone'



- ▶ 위에서는 's'와 't'는 그대로 사용하고, 'o'를 삽입하고, 'r'과 'o'를 각각 삭제
- ▶ 그리고 'n'을 그대로 사용하고, 마지막으로 'g'를 'e'로 대체하여, 총 4회의 편집 연산 수행

## 다른 시도

▶ 's'와 't'는 그대로 사용한 후, 'r'을 삭제하고, 'o'와 'n'을 그대로 사용한 후, 'g'를 'e'로 대체하여, 총 2회의 편집 연산만이 수행되고, 이는 최소 편집 횟수이다.



➤ S를 T로 바꾸는데 어떤 연산을 어느 문자에 수행하는가에 따라서 편집 연산 횟수가 달라진다.

# 부분 문제들을 어떻게 표현해야 할까?

- ➤ 접두부(prefix)를 살펴보자.
- > 'strong'⇒ 'stone'에 대해
- ▶ 예를 들어, 'stro'를 'sto'로 바꾸는 편집 거리를 <u>미리</u> <u>알면</u>, 'ng'를 'ne'로 바꾸는 편집 거리를 찾음으로써 주어진 입력에 대한 편집 거리를 구할 수 있다.

$$S = \begin{bmatrix} 1 & 2 & 3 & 4 \\ s & t & r & 0 \end{bmatrix} n g$$

$$T = \begin{bmatrix} s & t & 0 \\ 1 & 2 & 3 \end{bmatrix} n e$$

# 부분 문제 정의

> |S| = m, |T| = n  $S = s_1 s_2 s_3 s_4 \cdots s_m$  $T = t_1 t_2 t_3 t_4 \cdots t_n$ 

- ▶ E[i, j] = S의 처음 i개 문자를 T의 처음 j개 문자로 바꾸는데 필요한 편집 거리
- 'strong' → 'stone'에 대해서, 'stro'를 'sto'로 바꾸기 위한 편집 거리는 E[4, 3]이다.

	1	2	3	4	5	6
S	S	t	r	0	n	g
T	S	t	О	n	е	

# 'strong' ⇒ 'stone'에 대해

- $> s_{1\rightarrow}t_1$  ['s'  $\Rightarrow$  's']:  $s_1 = t_1 =$ 's'이므로 E[1, 1] = 0
- ▶ s<sub>1→</sub>t<sub>1</sub>t<sub>2</sub> ['s' ⇒ 'st']: s<sub>1</sub> = t<sub>1</sub> = 's' 이 고 , 't' 를 삽입하므로 E[1, 2] = 1
- >  $s_1 s_{2\rightarrow} t_1$  ['st'  $\Rightarrow$  's']:  $s_1 = t_1 =$  's'이고, 't'를 <u>삭제</u>하여 E[2, 1] = 1
- >  $s_1 s_2 t_1 t_2$  ['st'  $\Rightarrow$  'st']:  $s_1 = t_1 =$  's'이고,  $s_2 = t_2 =$  't'이므로 E[2, 2] = 0
  - 이 경우에는 E[1, 1] = 0이라는 결과를 이용하고 s<sub>2</sub> = t<sub>2</sub>=
     't'이므로, E[2, 2] = E[1,1] + 0 = 0

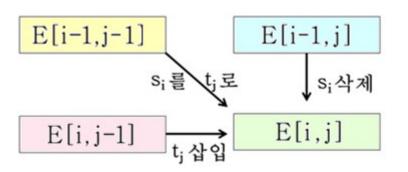
# E[4, 3]은 어떻게 계산하여야 할까?

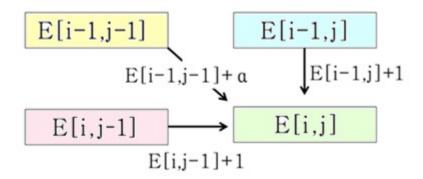
 $\gt$   $s_1 s_2 s_3 s_4 t_1 t_2 t_3$  ['stro'  $\Rightarrow$  'sto']

			T					
	E	900-	3	S	t	0		
		j	0	1	2	3		
	3	0	0	1	2	3		
	S	1	1	0	1	2		
S	t	2	2	1	0	1		
	r	3	3	2	1	1		
	O	4	4	3	2	1		

- $\triangleright$  E[4, 2]의 해를 알면,  $t_3$ ='o'를 <u>삽입</u>하면 E[4, 2] + 1
- $\triangleright$  E[3, 3]의 해를 알면,  $s_4$ ='o'를 <u>삭제</u>하면 E[3, 3] + 1
- $\triangleright$  E[3, 2]의 해를 알면,  $s_4$ ='o'과  $t_3$ ='o'이 같으므로 E[3, 2] +

## E[i, j]는?





$$E[i, j] = \min\{E[i, j-1]+1, E[i-1, j]+1, E[i-1, j-1]+\alpha\}$$
 
$$E[i, j] = \min\{E[i, j-1]+1, E[i-1, j]+1, E[i-1, j-1]+\alpha\}$$
 
$$E[i, j] = \min\{E[i, j-1]+1, E[i-1, j]+1, E[i-1, j-1]+\alpha\}$$

# 초기화

					,	Γ		
		9	3	$t_1$	$t_2$	$t_3$		$t_n$
			0	1	2	3		n
	3	0	0	1	2	3		n
	$S_1$	1	1		800 80			
	$S_2$	2	2					
S	<b>S</b> <sub>3</sub>	3	3				이 0	
		<u> </u>			ス	리리	-기 전	1에

m

- ○번 행이 0, 1, 2, …, n으로 초기화된 의미: S의 첫 문자를 처리하기 전에, 즉, S가 ε (공 문자열)인 상태에서 T의 문자를 좌에서 우로 하나씩 만들어 가는데 필요한 <u>삽입</u> 연산 횟수를 각각 나타낸다.
- ○번 열이 0, 1, 2, ···, m으로 초기화된 의미: 스트링 T를 ε
   로 만들기 위해서, S의 문자를 위에서 아래로 하나씩
   없애는데 필요한 삭제 연산 횟수를 각각 나타낸다.

 $S_{m}$ 

m

# 알고리즘

#### EditDistance

입력: 스트링 S, T, 단, S와 T의 길이는 각각 m과 n이다.

출력: S를 T로 변환하는 편집 거리, E[m, n]

- 1. for i=0 to m E[i, 0] = i // 0번 열의 초기화
- 2. for j=0 to n E[0, j] = j // 0번 행의 초기화
- 3. for i=1 to m
- 4. for j=1 to n
- 5.  $E[i, j] = min\{E[i, j-1] + 1, E[i-1, j] + 1, E[i-1, j-1] + \alpha\}$
- 6. return E[m, n]

# EditDistance 알고리즘 수행 과정

> 'strong'을 'stone'으로 바꾸는데 필요한 편집 거리

	Т	3	S	t	0	n	e
S		0	1	2	3	4	5
3	0	0	1	2	3	4	5
S	1	1	0	1	2	3	4
t	2	2	1	0	1	2	3
r	3	3	2	1	1	2	3
0	4	4	3	2	1	2	3
n	5	5	4	3	2	1	2
g	6	6	5	4	3	2	2

# E[1, 1]

 $E[1, 1] = \min\{E[1, 0] + 1, E[0, 1] + 1, E[0, 0] + \alpha\}$   $= \min\{(1+1), (1+1), (0+0)\}$  = 0

	T	3	S
S	ij	0	1
3	0	0 <	1
S	1	1	0

# E[1, 2]

 $E[2, 2] = \min\{E[2, 1] + 1, E[1, 2] + 1, E[1, 1] + \alpha\}$   $= \min\{(1+1), (1+1), (0+0)\}$  = 0

	T	3	S	t
S	ij	0	1	2
3	0	0	1	2
S	1	1	0	1
(t)	2	2	1	ŏŎ

# E[3, 2]

 $E[3, 2] = \min\{E[3, 1] + 1, E[2, 3] + 1, E[2, 2] + \alpha\}$   $= \min\{2 + 1, 0 + 1, 1 + 1\}$  = 1

	T	3	S	t
S	jį	0	1	2
3	0	0	1	2
S	1	1	0	1
t	2	2	1	Q
r	3	3	2 =	1

# E[4, 3]

$$E[4, 3] = \min\{E[4, 2] + 1, E[3, 3] + 1, E[3, 2] + \alpha\}$$

$$= \min\{(2+1), (1+1), (1+0)\}$$

$$= 1$$

	T	3	S	t	0
S	ij	0	1	2	3
3	0	0	1	2	3
S	1	1	0	1	2
t	2	2	1	0	1
r	3	3	2	1	1
0	4	4	3	2 =	1

## E[5, 4]

$$E[5, 4] = \min\{E[5, 3] + 1, E[4, 4] + 1, E[4, 3] + \alpha\}$$

$$= \min\{(2+1), (1+1), (1+0)\}$$

$$= 1$$

	T	3	S	t	O	n
S	ij	0	1	2	3	4
3	0	0	1	2	3	4
S	1	1	0	1	2	3
t	2	2	1	0	1	2
r	3	3	2	1	1	2
0	4	4	3	2	1	2
n	5	5	4	3	2 =	1

### E[6,5]

$$E[6, 5] = \min\{E[6, 4] + 1, E[5, 5] + 1, E[5, 4] + \alpha\}$$

$$= \min\{(2+1), (2+1), (1+1)\}$$

$$= 2$$

	T	3	S	t	0	n	e
S	ji	0	1	2	3	4	5
3	0	0	1	2	3	4	5
S	1	1	0	1	2	3	4
t	2	2	1	0	1	2	3
r	3	3	2	1	1	2	3
0	4	4	3	2	1	2	3
n	5	5	4	3	2	1	2
g	6	6	5	4	3	2 5	2

# 시간 복잡도

- ➤ EditDistance 알고리즘의 시간 복잡도는 O(mn). 단, m과 n은 두 스트링의 각각의 길이이다.
- ▶ 총 부분 문제의 수가 배열 E의 원소 수인 mxn이고, 각 부분 문제 (원소)를 계산하기 위해서 주위의 3 개의 부분 문제들의 해 (원소)를 참조한 후 최솟값을 찾는 것이므로 0(1) 시간 소요



#### **Applications**

- ➤ 2개의 스트링 사이의 편집 거리가 작으면, 이 스트링들이 서로 유사하다고 판단할 수 있으므로, 생물 정보 공학 (Bioinformatics) 및 의학 분야에서 두 개의 유전자가 얼마나 유사한가를 측정하는데 활용
  - 환자의 유전자 속에서 암 유전자와 유사한 유전자를 찾아내어 환자의 암을 미리 진단하는 연구와 암세포에만 있는 특징을 분석하여 항암제를 개발하는 연구에 활용되며, 좋은 형질을 가진 유전자들 탐색 등의 연구에도 활용
- ▶ 그 외에도 철자 오류 검색(Spell Checker), 광학 문자 인식 (Optical Character Recognition)에서의 보정 시스템 (Correction System), 자연어 번역(Natural Language Translation)소프트웨어 등에 활용

# 5.4 배낭 문제

- n개의 물건과 각 물건 i의 무게 w<sub>i</sub>와 가치 v<sub>i</sub>가 주어지고, 배낭의 용량이 C일 때, 배낭에 담을 수 있는 물건의 최대 가치는?
- ▶ 단, 배낭에 담은 물건의 무게의 합이 C를 초과하지 말아야 하고, 각 물건은 1개씩만 있다.
- ▶ 이러한 배낭 문제를 0-1 배낭 문제라고 한다.



# 부분 문제

- ▶ 문제에는 물건, 물건의 무게, 물건의 가치, 배낭의 용량, 모두 4가지의 요소가 있다.
- ▶ 물건과 물건의 무게는 부분 문제를 정의하는데 필요

K[i, w] = 물건 1~i까지만 고려하고, (임시) 배낭의 용량이 w일 때의 최대 가치 단, i = 1, 2, ..., n이고, w = 1, 2, 3, ..., C

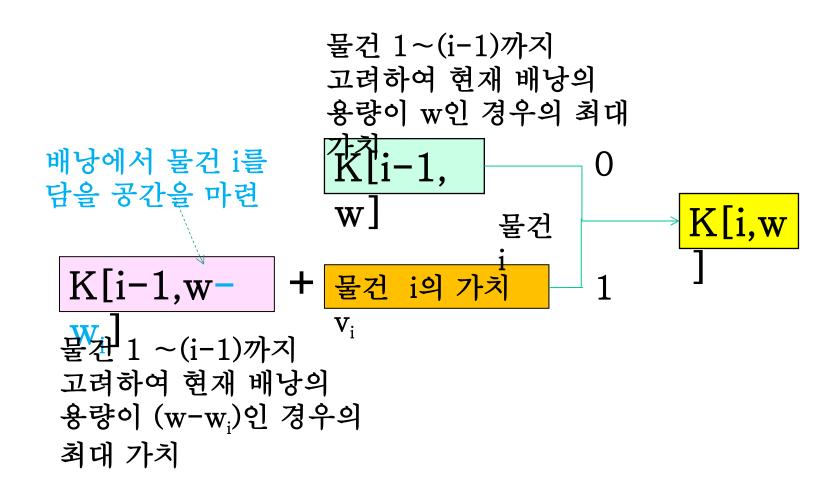
- 문제의 최적해 = K[n, C]

# 알고리즘

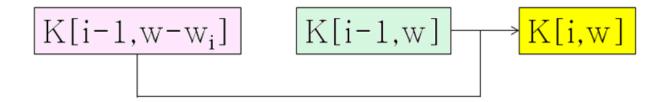
#### Knapsack

```
입력: 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 w;와 가치 v;,
    단, i = 1, 2, ···, n
출력: K[n, C]
1. for i = 0 to n K[i,0]=0 // 배낭의 용량이 0일 때
2. for w = 0 to C K[0,w]=0 // 물건 0이란 어떤 물건도 고려하지 않을 때
3. for i = 1 to n
4. for w = 1 to C // w는 배낭의 (임시) 용량
5.
      if (w<sub>i</sub> > w) // 물건 i의 무게가 임시 배낭 용량을 초과하면
       K[i, w] = K[i-1, w]
6.
7. else
                    // 물건 i를 배낭에 담지 않을 경우와 담을 경우 고려
       K[i, w] = max\{K[i-1, w], K[i-1, w-w_i] + v_i\}
8.
9. return K[n, C]
```

# Knapsack 알고리즘



# 부분 문제 간의 함축적 순서



# Knapsack 알고리즘 수행 과정

▶ 배낭의 용량 C=10kg

물건	1	2	3	4
무게 (kg)	5	4	6	3
가치(만원)	10	40	30	50



➤ Line 1~2: 0번 행과 0번 열의 각 원소를 0으로 초기화

C = 10

배낭 용량→ W=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	1	0										
4	40	2	0										
6	30	3	0										
3	50	4	0										

▶ w =1, 2, 3, 4일 때, 각각 물건 1을 담을 수 없다.
K[1, 1] = 0, K[1, 2] = 0, K[1, 3] = 0, K[1, 4] = 0



▶ W = 5 (배낭의 용량이 5kg)일 때,K[1, 5] = = 10





> w = 6, 7, 8, 9, 10일 때

K[1, 6] = K[1, 7] = K[1, 8] = K[1, 9] = K[1, 10] = 10





# 물건 1만 고려했을 때

C=10

배낭 용량→ W=			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건	0	0	0	0	0	0	0	0	0	0	0
5	10	i =1	0	0	0	0	0	10	10	10	10	10	10
4	40	2	0										
6	30	3	0										
3	50	4	0										

### 물건 2를 고려해보자

▶ w = 1, 2, 3 (배낭의 용량이 각각 1, 2, 3kg)일 때
 - K[2, 1] = 0, K[2, 2] = 0, K[2, 3] = 0



▶ w = 4 (배낭의 용량이 4kg)일 때K[2,4] = 40



▶ w = 5 (배낭의 용량이 5kg)일 때

$$K[2,5] = \max\{K[i-1,w], K[i-1,w-w_i] + v_i\}$$

- $= \max\{K[2-1,5], K[2-1,5-4]+40\}$
- $= \max\{K[1,5], K[1,1] + 40\}$
- $= \max\{10, 0+40\}$
- $= \max\{10, 40\} = 40$
- 물건 1을 배낭에서 빼낸 후, 물건 2를 담는다.
- 그 때의 가치가 40이다.



- ➤ w = 6, 7, 8일 때
  - 각각의 경우도 물건 1을 빼내고 물건 2를 배낭에 담는 것이 더 큰 가치를 얻는다.
  - K[2,6] = K[2,7] = K[2,8] = 40



➤ w = 9 (배낭의 용량이 9kg)일 때

$$K[2,9] = \max\{K[i-1,w], K[i-1,w-w_i] + v_i\}$$

 $= \max\{K[2-1,9], K[2-1,9-4]+40\}$ 

 $= \max\{K[1,9], K[1,5] + 40\}$ 

 $= \max\{10, 10+40\}$ 

 $= \max\{10, 50\} = 50$ 

• 물건 1, 2 둘 다를 담을 수 있다.



- ➤ w = 10 (배낭의 용량이 10kg)일 때
  - 물건 1, 2를 배낭에 둘 다 담을 수 있다.

C = 10

ŀ	배낭 용량→ W=					3	4	5	6	7	8	9	10
무게	가치 물건		0	0	0	0	0	0	0	0	0	0	0
5	10	1	0	0	0	0	0	10	10	10	10	10	10
4	40	i = 2	0	0	0	0	40	<b>40</b>	<b>40</b>	<b>40</b>	<b>40</b>	<b>50</b>	<b>50</b>
6	30	3	0										
3	50	4	0										

# 수행 결과

С

배낭 용량 → W=			0	1	2	3	4	5	6	7	8	9	10
물건	가치	무게	0	0	0	0	0	0	0	0	0	0	0
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

▶ 최적해 K[4, 10] = 90



## 시간 복잡도

- > 1개의 부분 문제에 대한 해를 구할 때의 시간 복잡도
  - line 5에서의 무게를 1번 비교한 후 line 6에서는 1 개의 부분 문제의 해를 참조하고, line 8에서는 2개의 해를 참조한 계산이므로 O(1) 시간
- ▶ 부분 문제의 수는 배열 K의 원소 수인 n x C
  - C = 배낭의 용량
- Knapsack 알고리즘의 시간 복잡도
  - $O(1) \times n \times C = O(nC)$



### **Applications**

- > 배낭 문제는 다양한 분야에서 의사 결정 과정에 활용
  - 원자재의 버리는 부분을 최소화 시키기 위한 자르기/분할
  - 금융 포트폴리오와 자산 투자의 선택
  - 암호 생성 시스템 (Merkle-Hellman Knapsack Cryptosystem) 등에 활용

## 5.5 동전 거스름돈 문제

- 대부분의 경우 그리디 알고리즘으로 해결되나, 해결 못하는 경우도 있다.
- ▶ DP 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.



## 부분 문제

- ▶ 문제에 주어진 요소들
  - 동전의 종류,  $d_1$ ,  $d_2$ ,  $\cdots$ ,  $d_k$ , 단,  $d_1 > d_2 > \cdots > d_k = 1$
  - 거스름돈 n원



배낭 문제의 DP 알고리즘에서 배낭의 용량을 1kg씩 증가시켜가며 문제를 해결

- ▶ 1원씩 증가시켜가며 문제를 해결하자.
- 거스름돈을 배낭의 용량과 같이 생각하자.

## 부분 문제

- ▶ 1차원 배열 C
  - C[1] = 1원을 거슬러 받을 때 사용되는 최소의 동전
     수
  - C[2] = 2원을 거슬러 받을 때 사용되는 최소의 동전
     수

:

- C[n] = n원을 거슬러 받을 때 사용되는 최소의 동전 수

#### C[j]를 계산하는데 어떤 부분 문제가 필요할까?

- ▶ 500원 동전이 필요하면 (j-500)원의 해, 즉, C[j-500]에다가 500원 동전 1개 추가
- ▶ 100원 동전이 필요하면 (j-100)원의 해, 즉, C[j-100]에다가 100원 동전 1개 추가
- ▶ 50원 동전이 필요하면 (j-50)원의 해, 즉, C[j-50]에다가 50원 동전 1개 추가
- ▶ 10원 동전이 필요하면 (j-10)원의 해, 즉, C[j-10]에다가 10원 동전 1개 추가
- ▶ 1원 동전이 필요하면 (j-1)원의 해, 즉, C[j-1]에다가 1원 동전 1개 추가

$$C[j] = \min_{1 \le i \le k} \{C[j-d_i] + 1\}, \text{ if } j \ge d_i$$

# 알고리즘

#### **DPCoinChange**

입력: 거스름돈 n원, k개의 동전의 액면,  $d_1 > d_2 > \cdots > d_k = 1$ 

출력: C[n]

- 1. for i = 1 to  $n C[i] = \infty$
- 2.C[0]=0
- 3. for j = 1 to n // j는 1원부터 증가하는 (임시) 거스름돈 액수
- 4. for i = 1 to k
- 5. if  $(d_i \le j)$  and  $(C[j-d_i]+1 < C[j])$
- 6.  $C[j] = C[j-d_i] + 1$
- 7. return C[n]

# DPCoinChange 알고리즘 수행 과정

 $\rightarrow$  d<sub>1</sub>=16, d<sub>2</sub>=10, d<sub>3</sub>=5, d<sub>4</sub>=1이고, 거스름돈 n=20일 때









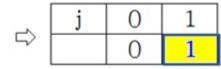
➤ Line 1~2: 배열 C 초기화

j	0	1	2	3	4	5	6	7	8	9	10	 16	17	18	19	20
С	0	$\infty$	8	 8	8	8	8	$\infty$								

## 거스름돈 1원 대원까지

ightharpoonup C[1] = C[j-1]+1 = C[1-1]+1 = C[0]+1 = 0+1 = 1

j	0	1
12 1 1 1 ()	0	$\infty$





ightharpoonup C[2] = C[j-1]+1 = C[2-1]+1 = C[1]+1 = 1+1 = 2

j	1	2
	1	$\infty$



j	1	2
	1	2



ightharpoonup C[3] = C[j-1]+1 = C[3-1]+1 = C[2]+1 = 2+1 = 3

j	2	3
	2	$\infty$

$$\Rightarrow$$

j	2	3
	2	3







 $\triangleright$  C[4] = C[j-1]+1 = C[4-1]+1 = C[3]+1 = 3+1 = 4

j	3	4
	3	8

$$\Rightarrow$$

j	3	4
	3	4

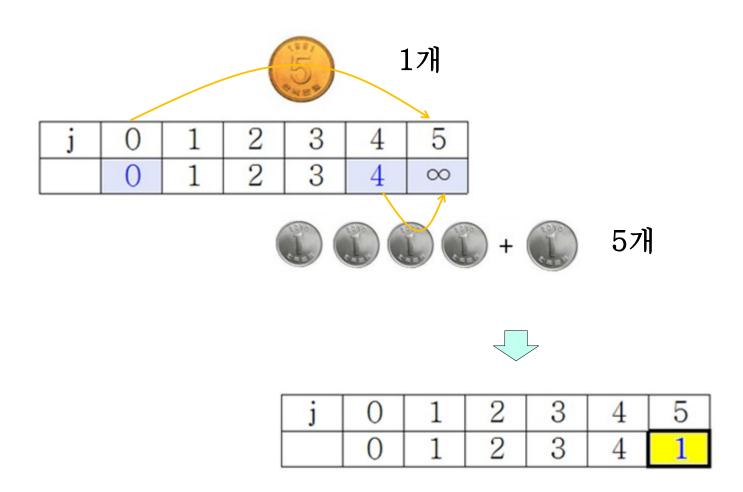








# 거스름돈 5원



# 거스름돈 6, 7, 8, 9원



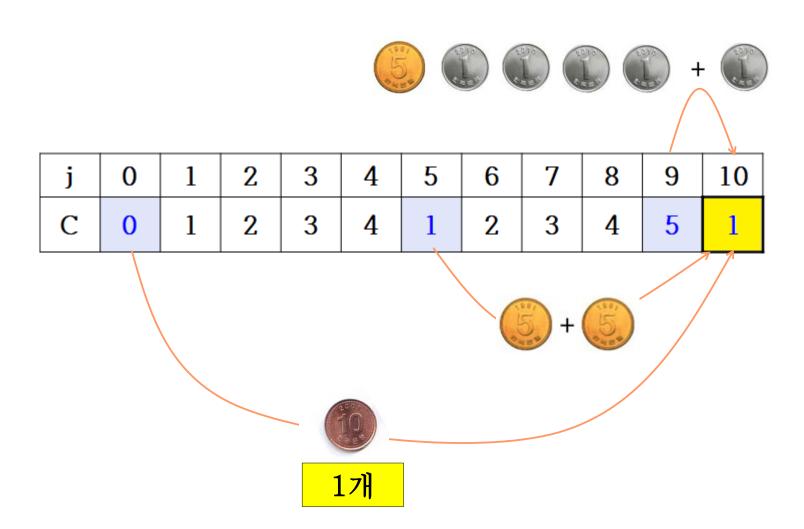
+ <del>5</del> 37H

(1) (1) + (5) 47H

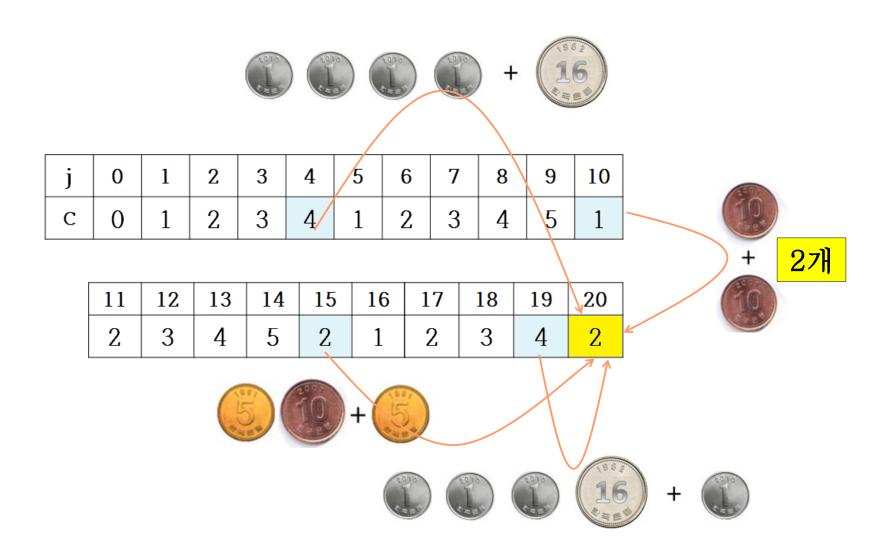
(1) (1) (2) + (5) 5개

j	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	1	$\infty$	$\infty$	$\infty$	$\infty$
	0	1	2	3	4	1	2	$\infty$	<sub>∞</sub>	$\infty$
C	0	1	2	3	4	1	2	3	<sub>∞</sub>	$\infty$
	0	1	2	3	4	1	2	3	4	$\infty$
	0	1	2	3	4	1	2	3	4	5

# 거스름돈 10원



# 거스름돈 20원



## 수행 결과

▶ 거스름돈 20원에 대한 최종해 = C[20]=2

▶ 그리디 알고리즘은 20원에 대해 16원 동전을 먼저 '욕심 내어' 취하고, 4원이 남게 되어, 1원 4개를 취하여, 모두 5개의 동전이 해라고 답한다.



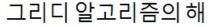








( a st







동적 계획 알고리즘의 해

## 시간 복잡도

- -0(nk)
- 거스름돈 j가 1원□h원까지 변하며, 각각의 j에 대해서 최대 모든 동전 (d₁, d₂, ···, d㎏)을 (즉, k개를) 1번씩 고려하기 때문



➤ 동적 계획(Dynamic Programming) 알고리즘은 최적화 문제를 해결하는 알고리즘으로서 입력 크기가 작은 부분 문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여, 주어진 입력의 문제를 해결하는 알고리즘

▶ DP 알고리즘에는 부분 문제들 사이에 함축적 순서가 존재한다.



- ➤ 모든 쌍 최단 경로(All Pairs Shortest Paths) 문제를 위한 Floyd-Warshall 알고리즘은 O(n³) 시간에 해를 찾는다.
  - 경유 가능한 점들을 점 1로부터 시작하여, 점 1과 2, 그다음엔 점 1, 2, 3으로 하나씩 추가하여, 마지막에는 점 1에서 점 n까지의 모든 점을 경유 가능한 점들로 고려하면서, 모든 쌍의 최단 경로의 거리를 계산한다.
- ▶ 연속 행렬 곱셈(Chained Matrix Multiplications) 문제를 위한 O(n³) DP 알고리즘은 이웃하는 행렬들끼리 곱하는 모든 부분 문제들을 해결하여 최적해를 찾는다.



- ▶ 편집 거리(Edit Distance) 문제를 위한 DP 알고리즘은 E[i, j]를 3개의 부분 문제 E[i, j-1], E[i-1, j], E[i-1, j-1]만을 참조하여 계산한다. 시간 복잡도는 O(mn)이다. 단, m과 n은 두 스트링의 길이이다.
- ▶ 배낭(Knapsack) 문제를 위한 DP 알고리즘은 부분 문제 K[i, w]를 물건 1~i까지만 고려하고, (임시) 배낭의 용량이 w일 때의 최대 가치로 정의하여, i를 1 ~ 물건 수인 n까지, w를 1 ~ 배낭 용량 C까지 변화시키며 해를 찾는다. 시간 복잡도는 0(nC)이다.



- ➤ 동전 거스름돈(Coin Change)문제는 1원씩 증가시켜 문제를 해결한다. 시간 복잡도는 0(nk)이다. 단, n은 거스름돈 액수이고, k는 동전 종류의 수이다.
- ▶ DP 알고리즘은 부분 문제들 사이의 관계를 빠짐없이 고려하여 문제를 해결한다.
- ▶ DP 알고리즘은 최적 부분 구조 (optimal substructure) 또는 최적성 원칙 (principle of optimality) 특성을 가지고 있다.
  - 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있다.
  - 그리디 알고리즘도 같은 속성을 가진다.