

Chapter 5. 동적 계획 알고리즘

동적 계획 알고리즘

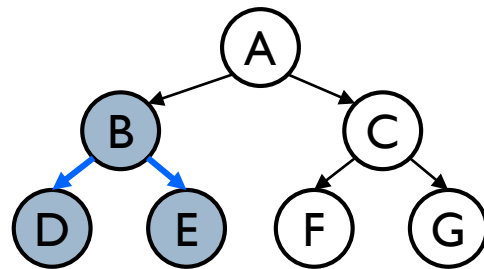
- ▶ 동적 계획(Dynamic Programming) 알고리즘은 그리디 알고리즘과 같이 최적화 문제를 해결하는 알고리즘이다.
- ▶ 동적 계획 알고리즘은 먼저 입력 크기가 작은 부분문제들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분문제들을 해결하여, 최종적으로 원래 주어진 입력의 문제를 해결하는 알고리즘이다.

가장 작은 것부터

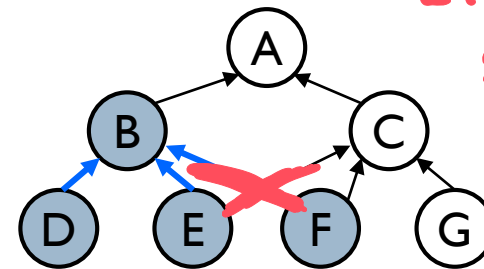
동적 계획 알고리즘

왜 쓰는지 /

- ▶ 분할 정복 알고리즘과 동적 계획 알고리즘의 전형적인 부분문제들 사이의 관계



분할 정복 알고리즘



동적 계획 알고리즘

함수 재사용 느낌
원시적인 느낌

- ▶ 분할 정복 알고리즘의 부분문제들 사이의 관계
 - ▶ A는 B와 C로 분할되고, B는 D와 E로 분할된다. 단, D, E, F, G는 각각 더 이상 분할할 수 없는(또는 가장 작은 크기의) 부분문제들이다.
 - ▶ D와 E의 해를 취합하여 B의 해를 구하고, F와 G의 해를 취합하여 C의 해를 구한다. 마지막으로 B와 C의 해를 취합하여 A의 해를 구한다.
 - ▶ 분할 정복 알고리즘은 부분문제의 해를 중복 사용하지 않는다.

가장 작은 단위들부터

동적 계획 알고리즘

동적 계획 알고리즘

- ▶ 동적 계획 알고리즘의 부분문제들 사이의 관계
 - ▶ 동적 계획 알고리즘은 먼저 최소 단위의 부분문제 D, E, F, G의 해를 각각 구한다.
 - ▶ D, E, F의 해를 이용하여 B의 해를 구하고, E, F, G의 해를 이용하여 C의 해를 구한다.
 - ▶ E와 F는 B와 C의 해를 구하는데 중복 사용한다.
 - ▶ B와 C의 해를 구하는데 E와 F의 해 모두를 이용한다.
- ▶ 동적 계획 알고리즘에는 부분문제들 사이에 의존적 관계가 존재한다.
 - ▶ 예를 들면, D, E, F의 해가 B를 해결하는데 사용되는 관계가 있다.
 - ▶ 이러한 관계는 문제 또는 입력에 따라 다르고, 대부분의 경우 뚜렷이 보이지 않아서 '함축적인 순서(implicit order)'라고 한다.

5.1 모든 쌍 최단 경로

- ▶ 모든 쌍 최단 경로(All Pairs Shortest Paths) 문제는 각 쌍의 점 사이의 최단 경로를 찾는 문제이다. 서울 각과걸 43로 있다

	서울	인천	수원	대전	전주	광주	대구	울산	부산
서울	○	40	41	154	232	320	297	408	432
인천	ex) 45	○	55	174	253	352	318	447	453
수원			○	133	189	300	268	356	391
대전				○	97	185	149	259	283
전주					○	106	220	331	323
광주						○	219	330	268
대구							○	111	136
울산								○	53
부산									○

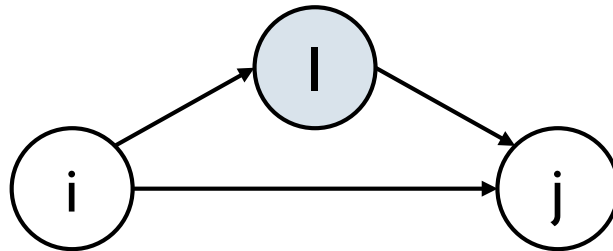


모든 쌍 최단 경로

- ▶ 문제를 해결하려면, 각 점을 시작점으로 정하여 **다익스트라 알고리즘**을 수행하면 된다.
 - ▶ 이때의 시간복잡도는 배열을 사용하면 $(n-1) \times O(n^2) = O(n^3)$ 이다. 단, n 은 점의 수이다.
- ▶ 모든 쌍 최단 경로를 찾는 동적 계획 알고리즘을 **플로이드-워셜 알고리즘** 또는 **플로이드 알고리즘**이라고 한다.
 - ▶ 워셜(Warshall)은 그래프에서 모든 쌍의 경로 존재 여부(transitive closure)를 찾아내는 동적 계획 알고리즘을 제안했고, 플로이드(Floyd)는 이를 변형하여 모든 쌍 최단 경로를 찾는 알고리즘을 고안하였다.
 - ▶ 플로이드 알고리즘의 시간복잡도는 $O(n^3)$ 으로 다익스트라 알고리즘을 $(n-1)$ 번 사용할 때의 시간복잡도와 동일하나, 매우 간단하여 다익스트라 알고리즘을 사용하는 것보다 효율적이다.

모든 쌍 최단 경로: 핵심 아이디어

- ▶ 동적 계획 알고리즘으로 모든 쌍 최단 경로 문제를 해결하려면 먼저 부분문제들을 찾아야 한다. 이를 위해 일단 그래프의 점의 수가 적을 때를 생각해보자.
- ▶ 그래프에 3개의 점이 있는 경우, 점 i 에서 점 j 까지의 최단 경로를 찾으려면 2가지 경로, 즉 점 i 에서 점 j 로 직접 가는 경로와 점 l 을 경유하는 경로 중에서 짧은 것을 선택하면 된다.



모든 쌍 최단 경로: 핵심 아이디어

▶ 경유 가능한 점들은 하나씩 늘어간다.

- ▶ 점 1로부터 시작하여, 점 1과 2, 그 다음에는 점 1, 2, 3으로 하나씩 추가하여, 마지막에는 점 1에서 n 까지의 모든 점을 경유 가능한 점들로 고려하면서, 모든 쌍의 최단 경로의 거리를 계산하는 것이다.

점 1만 경유 | 점 1, 2만 경유 | 점 1, 2, 3만 경유
이미지 안에서 만 계산

▶ 부분문제 정의

- ▶ 단, 입력 그래프의 점을 각각 $1, 2, 3, \dots, n$ 이라 하자.

D_{ij}^k = 점 $\{1, 2, \dots, k\}$ 만을 경유가능한 점들로 고려하여, 점 i 로부터 점 j 까지의 모든 경로 중에서 가장 짧은 경로의 거리

distance $i-j$ 점 n 개까지
 $1 \sim 3$ 경유가능한

모든 쌍 최단 경로: 핵심 아이디어

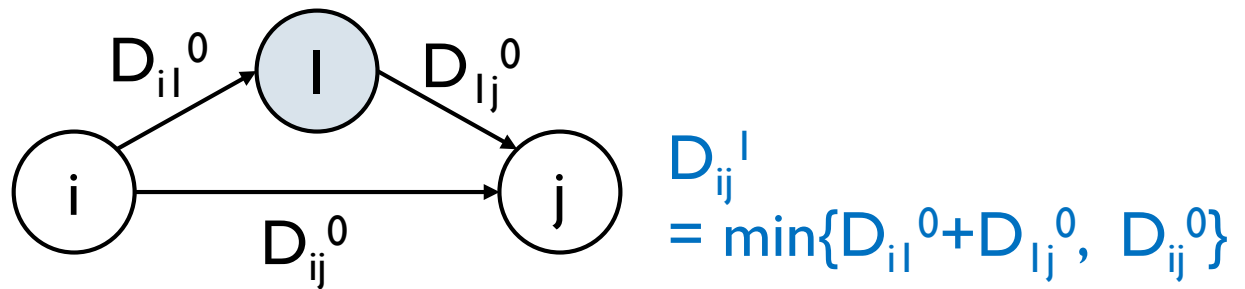
▶ 부분문제 D_{ij}^k

- ▶ 주의할 것은 점 i 에서 점 k 까지의 모든 점들을 반드시 경유하는 경로를 의미하는 것이 아니다.
- ▶ 심지어는 D_{ij}^k 는 이 점들을 하나도 경유하지 않으면서 점 i 에서 점 j 에 도달하는 경로, 즉 선분 (i, j) 가 최단 경로가 될 수도 있다.
- ▶ 여기서 $k \neq i, k \neq j$ 이고, $k=0$ 인 경우, 점 0 은 그래프에 없으므로 어떤 점도 경유하지 않는다는 것을 의미한다. 따라서 D_{ij}^0 은 입력으로 주어지는 선분 (i, j) 의 가중치이다.

0 간접 다이렉트

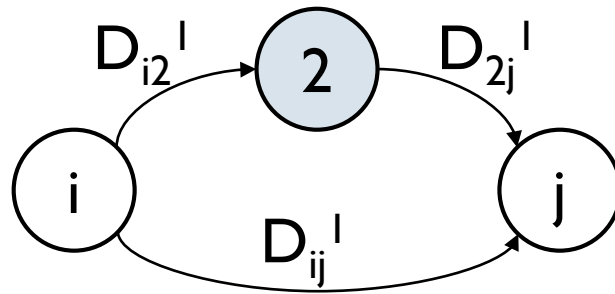
모든 쌍 최단 경로: 핵심 아이디어

- ▶ D_{ij}^l 은 i 에서 점 l 을 경유하여 j 로 가는 경로와 i 에서 j 로 직접 가는 경로, 즉 선분(i, j) 중에서 짧은 거리이다. 따라서 모든 쌍 i 와 j 에 대하여 D_{ij}^l 을 계산하는 것이 가장 작은 부분문제들이다. 단, $i \neq l, j \neq l$ 이다.



모든 쌍 최단 경로: 핵심 아이디어

- ▶ 그 다음에는 i 에서 점 2를 경유하여 j 로 가는 경로의 거리와 D_{ij}^1 중에서 더 짧은 거리를 D_{ij}^2 로 정한다. 단, 점 2를 경유하는 경로의 거리는 $D_{i2}^1 + D_{2j}^1$ 이다. 모든 쌍 i 와 j 에 대하여 D_{ij}^2 를 계산하는 것이 그 다음으로 큰 부분문제들이다. 단, $i \neq 2$, $j \neq 2$ 이다.

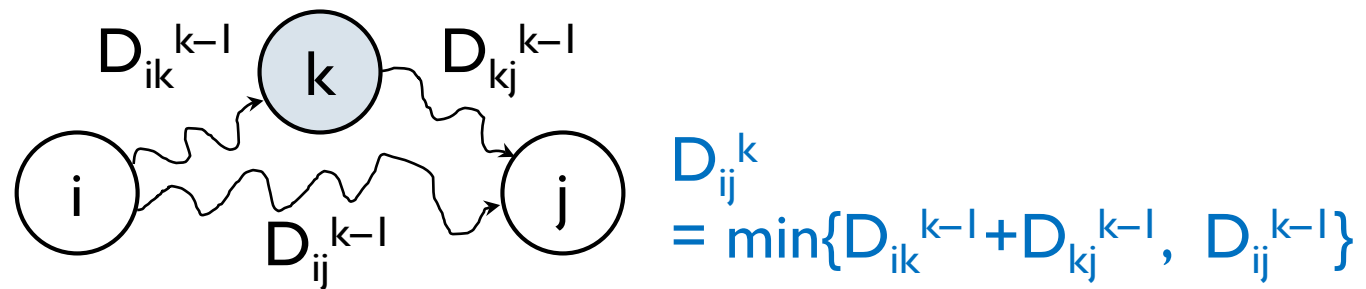


$$D_{ij}^2 = \min\{D_{i2}^1 + D_{2j}^1, D_{ij}^1\}$$

$$D_{ij}^3 = \min(D_{i3}^2 + D_{3j}^2, D_{ij}^2)$$

모든 쌍 최단 경로: 핵심 아이디어

- ▶ 점 i 에서 점 k 를 경유하여 j 로 가는 경로의 거리와 D_{ij}^{k-1} 중에서 짧은 것을 경로로 정한다. 단, 점 k 를 경유하는 경로의 거리는 $D_{ik}^{k-1} + D_{kj}^{k-1}$ 이고, $i \neq k, j \neq k$ 이다.



- ▶ 이런 방식으로 k 가 1에서 n 이 될 때까지 D_{ij}^k 를 계산해서 D_{ij}^n , 즉 모든 점을 경유가능한 점들로 고려된 모든 쌍 i 와 j 의 최단 경로의 거리를 찾는 방식이 플로이드의 모든 쌍 최단 경로 알고리즘이다.

모든 쌍 최단 경로 알고리즘

AllPairsShortest

입력: 2차원 배열 D, 단, $D[i, j]$ = 선분 (i, j) 의 가중치, 만일 선분 (i, j) 가 존재하지 않으면 $D[i, j] = \infty$ 이고, 모든 i 에 대하여 $D[i, i] = 0$ 이다.

출력: 모든 쌍 최단 경로의 거리를 저장한 2차원 배열 D

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

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

경로는 추가는 X

$$D[1, 6] = \min\{D[1, 2] + D[2, 6], D[1, 6]\}$$

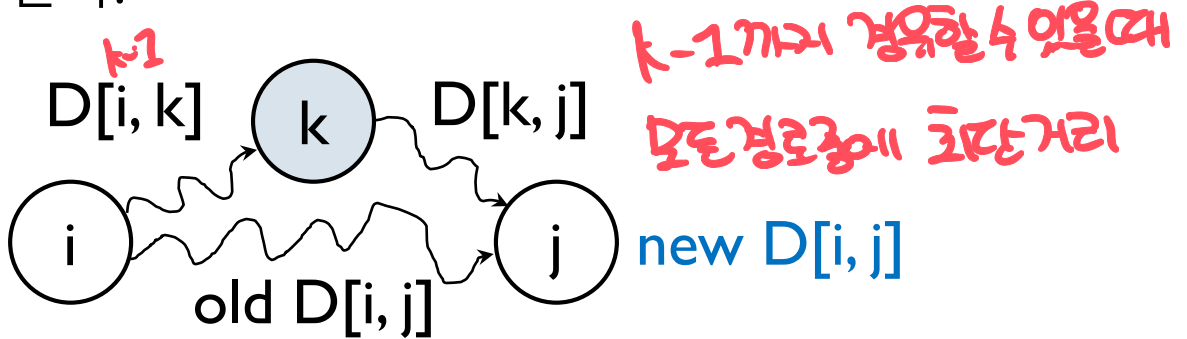
경로는 이전 100에서 다할때는

j는 k, 1을 두지 않음

$$k=2, i=1$$

$$j=4와 j=6에너$$

모든 쌍 최단 경로 알고리즘

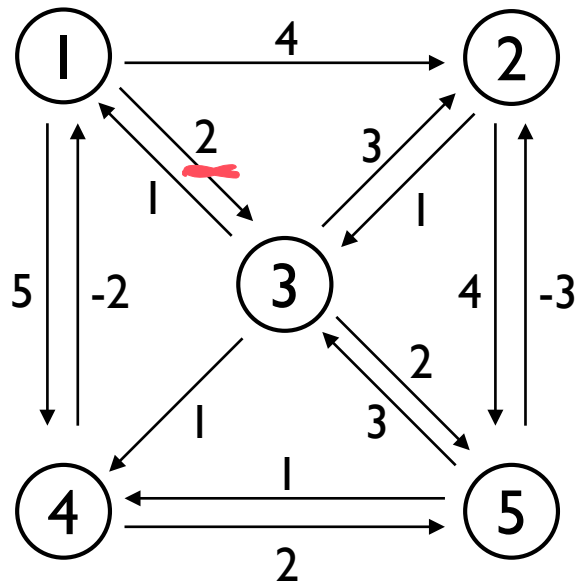
Line 1	for-루프는 k 가 1에서 n 까지 변하는데, 이는 경유 가능한 점을 1부터 n 까지 확장시키기 위한 것이다.
Line 2~3	점들의 각 쌍, 즉 $1-1, 1-2, 1-3, \dots, 1-n, 2-1, 2-2, \dots, 2-n, \dots, n-1, n-2, \dots, n-n$ 을 하나씩 고려하기 위한 루프이다. 단, $i=j$ 또는 $i=k$ 또는 $j=k$ 의 경우에는 수행하지 않는다.
Line 4	<p>각 점의 쌍 $i-j$에 대해 i에서 j까지의 거리가 k를 포함하여 경유하는 경로의 거리, 즉 $D[i, k] + D[k, j]$와 점 $\{1, 2, \dots, (k-1)\}$만을 경유 가능한 점들로 고려하여 계산된 최단 경로의 거리 $D[i, j]$ 중에서 짧은 거리를 $D[i, j]$로 갱신한다.</p> 

모든 쌍 최단 경로 알고리즘

▶ AllPairsShortest 알고리즘

- ▶ 모든 쌍 최단 경로 문제의 부분문제 간의 함축적 순서는 line 4에 표현되어 있다. 즉, 새로운 $D[i, j]$ 를 계산하기 위해서 미리 계산되어 있어야 할 부분문제들은 $D[i, k]$ 와 $D[k, j]$ 이다.
 - ▶ $D[i, j] = \min\{D[i, k] + D[k, j], D[i, j]\}$
 - ▶ $D_{ij}^k = \min\{D_{ik}^{k-1} + D_{kj}^{k-1}, D_{ij}^{k-1}\}$
- ▶ AllPairsShortest 알고리즘의 입력 그래프에는 사이클상의 선분들의 가중치 합이 음수가 되는 사이클은 없어야 한다. 이러한 사이클을 음수 사이클(negative cycle)이라 하는데, 최단 경로를 찾을 때 음수 사이클이 있으면, 이 사이클을 반복하여 지나갈 때마다 경로의 거리가 감소되기 때문이다.

AllPairsShortest 예제



D_{ij}^0

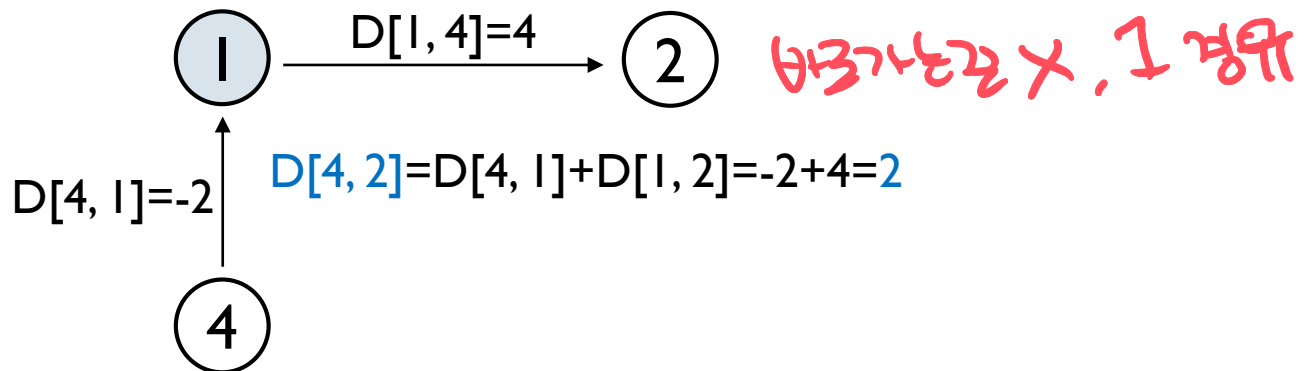
D	1	2	3	4	5
1	0	4	2	5	∞
2	∞	0	1	∞	4
3	1	3	0	1	2
4	-2	∞	∞	0	2
5	∞	-3	3	1	0

- 배열 D 의 원소들이 k 가 1부터 5까지 증가함에 따라서 갱신되는 것을 살펴보자.

AllPairsShortest 예제

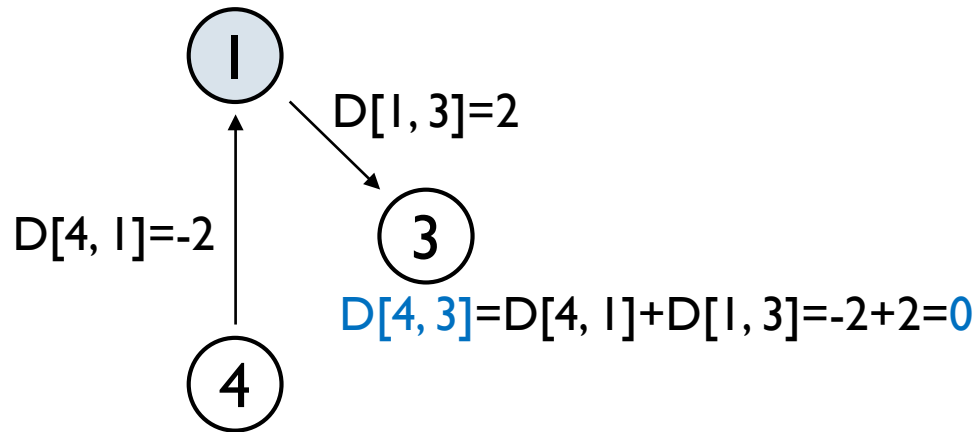
▶ $k=1$ 일 때:

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



AllPairsShortest 예제

- ▶ $D[4, 3] = \min\{D[4, 3], D[4, 1] + D[1, 3]\} = \min\{\infty, -2 + 2\} = 0$ // 갱신됨



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

AllPairsShortest 예제

- ▶ $k=1$ 일 때 $D[4, 2], D[4, 3]$ 이 각각 2, 0으로 갱신된다. 다른 원소들은 변하지 않았다.

D_{ij}^1

1경로

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

AllPairsShortest 예제

▶ k=2일 때

- ▶ $D[1, 5]$ 가 $1 \rightarrow 2 \rightarrow 5$ 의 거리인 8로 갱신된다.
- ▶ $D[5, 3]$ 이 $5 \rightarrow 2 \rightarrow 3$ 의 거리인 -2로 갱신된다.

D_{ij}^2

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

도약.

4

4

도약

-3, 1

AllPairsShortest 예제

- ▶ $k=3$ 일 때 총 7개의 원소가 갱신된다.

$$D_{ij}^3$$

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

AllPairsShortest 예제

- ▶ $k=4$ 일 때 총 3개의 원소가 갱신된다.

$$D_{ij}^4$$

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

AllPairsShortest 예제

- ▶ $k=5$ 일 때 총 3개의 원소가 갱신되고, 이것이 주어진 입력에 대한 최종해이다.

$$D_{ij}^5$$

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

시간복잡도

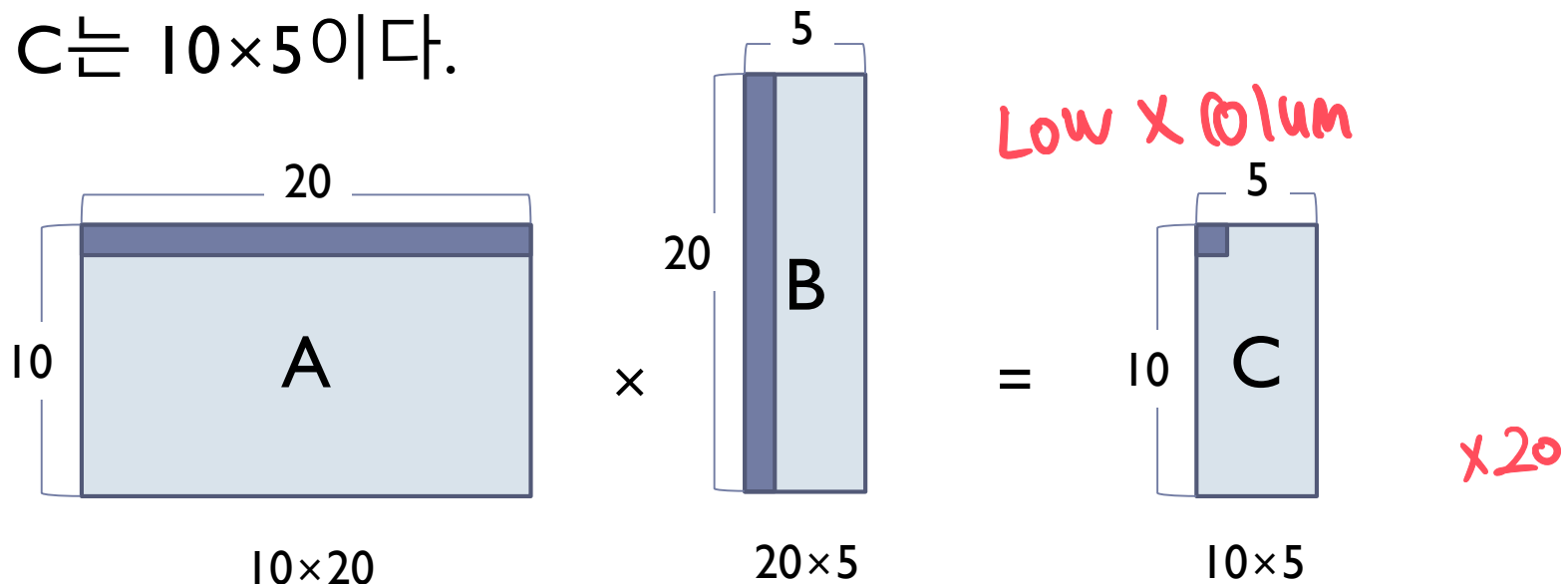
- ▶ AllPairsShortest의 시간복잡도는 위의 예제에서 보았듯이 각 k 에 대해서 모든 i, j 쌍에 대해 계산되므로, 총 $n \times n \times n = n^3$ 회 계산이 이루어지고, 각 계산은 $O(1)$ 시간이 걸린다.
- ▶ 따라서 AllPairsShortest의 시간복잡도는 $O(n^3)$ 이다.

응용

- ▶ 구글(Google) 등 웹사이트의 지도 서비스
- ▶ 자동차 네비게이션 서비스
- ▶ 지리 정보 시스템(GIS)에서의 네트워크 분석
- ▶ 통신 네트워크와 모바일 통신 분야
- ▶ 게임
- ▶ 산업 공학, 경영 공학의 운영 연구(Operations Research)
- ▶ 로봇 공학
- ▶ 교통 공학
- ▶ VLSI 디자인 분야 등

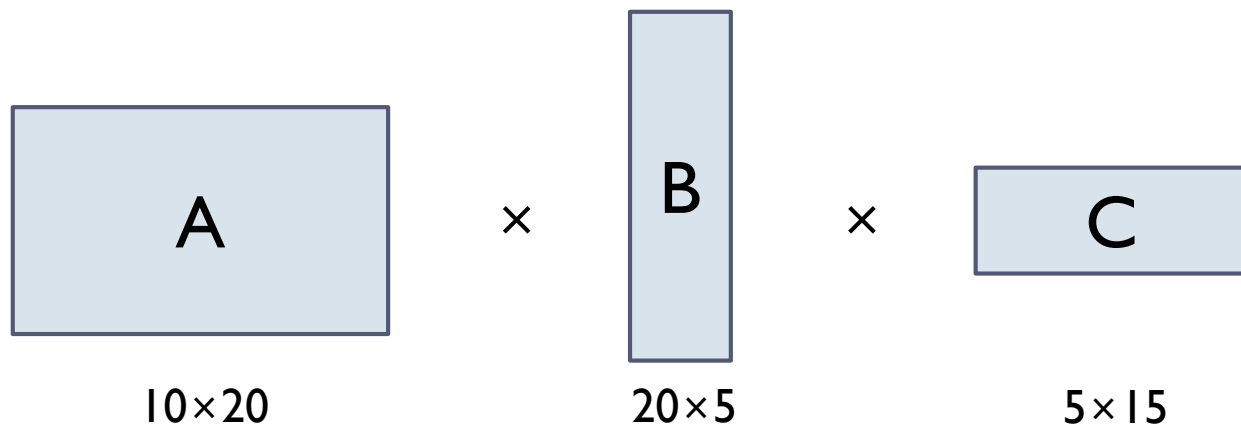
5.2 연속 행렬 곱셈

- ▶ 연속 행렬 곱셈(Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소 간의 최소 곱셈 횟수를 찾는 문제이다.
- ▶ 10×20 행렬 A와 20×5 행렬 B를 곱하는데 원소 간의 곱셈 횟수는 $10 \times 20 \times 5 = 1000$ 이다. 그리고 두 행렬을 곱한 결과 행렬 C는 10×5 이다.



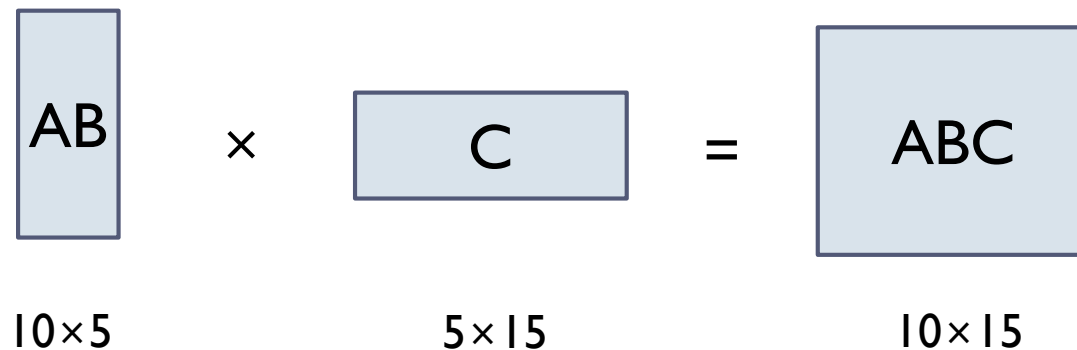
연속 행렬 곱셈

- ▶ 행렬 C의 1개의 원소를 위해서 행렬 A의 1개의 행에 있는 20개 원소와 행렬 B의 1개의 열에 있는 20개의 원소를 각각 곱한 값을 더해야 하므로 20회의 곱셈이 필요하다.
- ▶ 3개의 행렬을 곱해야 하는 경우
 - ▶ 연속된 행렬의 곱셈에는 결합 법칙이 허용된다.
 - ▶ 즉, $A \times B \times C = (A \times B) \times C = A \times (B \times C)$ 이다.
 - ▶ 다음과 같이 행렬 A가 10×20 , 행렬 B가 20×5 , 행렬 C가 5×15 라고 하자.



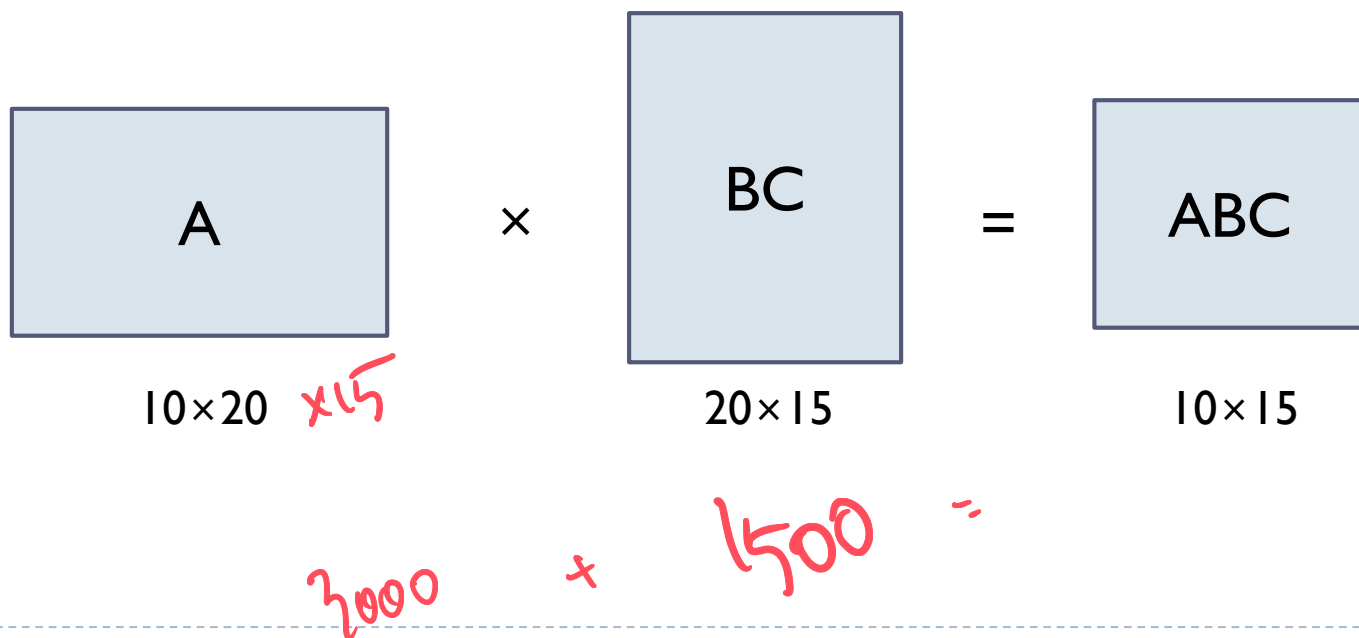
연속 행렬 곱셈

- ▶ 먼저 $A \times B$ 를 계산한 후에 그 결과 행렬과 행렬 C 를 곱하기 위한 원소 간의 곱셈 횟수를 세어 보면, $A \times B$ 를 계산하는데 $10 \times 20 \times 5 = 1000$ 번의 곱셈이 필요하고, 그 결과 행렬의 크기가 10×5 이므로, 이에 행렬 C 를 곱하는데 $10 \times 5 \times 15 = 750$ 번의 곱셈이 필요하다.
- ▶ 총 $1000 + 750 = 1750$ 회의 원소의 곱셈이 필요하다.



연속 행렬 곱셈

- ▶ 이번에는 $B \times C$ 를 먼저 계산한 후에 행렬 A를 그 결과 행렬과 곱하면, $B \times C$ 를 계산하는데 $20 \times 5 \times 15 = 1500$ 번의 곱셈이 필요하고, 그 결과 20×15 행렬이 만들어지므로, 이를 행렬 A와 곱하는데 $10 \times 20 \times 15 = 3000$ 번의 곱셈이 필요하다.
- ▶ 총 $1500 + 3000 = 4500$ 회의 곱셈이 필요하다.



연속 행렬 곱셈

- ▶ 동일한 결과를 얻음에도 불구하고 원소 간의 곱셈 횟수가 $4500 - 1750 = 2750$ 이나 차이가 난다.
 - ▶ 따라서, 연속된 행렬을 곱하는데 필요한 원소 간의 곱셈 횟수를 최소화시키기 위해서는 적절한 행렬의 곱셈 순서를 찾아야 한다.
- ▶ 연속 행렬 곱셈의 부분문제
 - ▶ 행렬의 곱셈은 교환법칙이 성립하지 않으므로 주어진 행렬의 순서를 지켜서 이웃하는 행렬끼리 곱해야 한다.
 - ▶ $A \times B \times C \times D \times E$ 를 계산하는데, B 를 건너뛰어서 $A \times C$ 를 수행한다든지, B 와 C 를 건너뛰어서 $A \times D$ 를 먼저 수행할 수 없다. 따라서 $A \times B \times C \times D \times E$ 의 부분문제는 다음과 같이 만들어진다.

$A \times B \times C \times D \times E$ 의 부분문제

부분문제 크기						부분문제 개수
1	A	B	C	D	E	5개
2	$A \times B$	$B \times C$	$C \times D$	$D \times E$		4개

- ▶ 맨 윗줄의 가장 작은 부분문제들은 입력으로 주어진 각각의 행렬이고, 크기가 2인 부분문제는 2개의 이웃하는 행렬의 곱셈으로 이루어진 4개이다. 여기서 부분문제들이 겹쳐져 있는 것을 알 수 있다.

$A \times B \times C \times D \times E$ 의 부분문제

부분문제 크기		부분문제 개수
3	$A \times B \times C$ $B \times C \times D$ $C \times D \times E$	3개
4	$A \times B \times C \times D$ $B \times C \times D \times E$	2개
5	$A \times B \times C \times D \times E$	1개

- ▶ 그 다음은 크기가 3인 부분문제가 3개이고, 이들 역시 서로 이웃하는 부분문제들끼리 겹쳐 있음을 확인할 수 있다.
- ▶ 다음 줄에는 크기가 4인 부분문제가 2개이고, 마지막에는 1개의 문제로서 입력으로 주어진 문제이다.

연속 행렬 곱셈 알고리즘

MatrixChain

입력: 연속된 행렬 $A_1 \times A_2 \times \dots \times A_n$, 단, A_1 은 $d_0 \times d_1$, A_2 는 $d_1 \times d_2$, ..., A_n 은 $d_{n-1} \times d_n$ 이다.
출력: 입력의 행렬 곱셈에 필요한 원소 간의 최소 곱셈 횟수

```

1 for i = 1 to n
2   C[i, i] = 0
3   for L = 1 to n-1 { // L은 부분문제의 크기를 조절하는 인덱스이다.
4     for i = 1 to n-L {
5       j = i + L
6       C[i, j] = ∞
7       for k = i to j-1 {
8         temp = C[i, k] + C[k+1, j] + d[i-1]d_kd_j
9         if (temp < C[i, j])
10            C[i, j] = temp
11       }
12     }
13   }
14 return C[1, n]

```

Handwritten notes in Korean:

- 12. 23. 34. 45 (referring to indices)
- $C[i, j]$
- $A_i \times \dots \times A_j$
- $A_{n-1} \times A_n$
- L (referring to the loop variable)
- $A_1 d_{i-1} \times d_i$
- $(A_i) \times (A_{i+1} \times \dots \times A_j)$
- $(A_i \times A_{i+1}) \times \dots$
- $(A_i \times \dots \times A_{j-1}) \times A_j$
- 이러한 계산은 L을 증가시키면서 반복적으로 계산하는 것 (This calculation is repeated by increasing L)
- 이러한 계산은 L을 증가시키면서 반복적으로 계산하는 것 (This calculation is repeated by increasing L)

연속 행렬 곱셈 알고리즘

Line 1~2	배열의 대각선 원소들, 즉, $C[1, 1], C[2, 2], \dots, C[n, n]$ 을 0으로 각각 초기화시킨다. 그 의미는 행렬 A_1, A_2, \dots, A_n 을 각각 계산하는데 필요한 원소 간의 곱셈 횟수가 0이란 뜻이다. $C[i, i]$ 는 가장 작은 부분문제의 해이다.
Line 3~4	for-루프의 L 은 1부터 $(n-1)$ 까지 변하는데, L 은 부분문제의 크기를 2부터 n 까지 조절하기 위한 변수이다. 즉, 이를 위해 line 4의 for-루프의 i 가 1부터 $(n-L)$ 까지 변한다. <ul style="list-style-type: none">• $L=1$일 때, i는 1부터 $(n-1)$까지 변하므로, 크기가 2인 부분문제의 수가 $(n-1)$개이다.• $L=2$일 때, i는 1부터 $(n-2)$까지 변하므로, 크기가 3인 부분문제의 수가 $(n-2)$개이다.• $L=3$일 때, i는 1부터 $(n-3)$까지 변하므로, 크기가 4인 부분문제의 수가 $(n-3)$개이다.

연속 행렬 곱셈 알고리즘

L=1

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-1)개

L=2

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-2)개

L=3

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-3)개

연속 행렬 곱셈 알고리즘

Line 3~4

- $L=n-2$ 일 때, i 는 1부터 $n-(n-2) = 2$ 까지 변하므로, 크기가 $(n-1)$ 인 부분문제의 수는 2개이다.
- $L=n-1$ 일 때, i 는 1부터 $n-(n-1) = 1$ 까지 변하므로, 크기가 n 인 부분문제의 수는 1개이다.

$L = n-2$

C	1	2	3	.	.	$n-1$	n
1	0						
2		0					
3			0				
.				0			
.					0		
$n-1$						0	
n							0

2개

$L = n-1$

C	1	2	3	.	.	$n-1$	n
1	0						
2		0					
3			0				
.				0			
.					0		
$n-1$						0	
n							0

1개

연속 행렬 곱셈 알고리즘

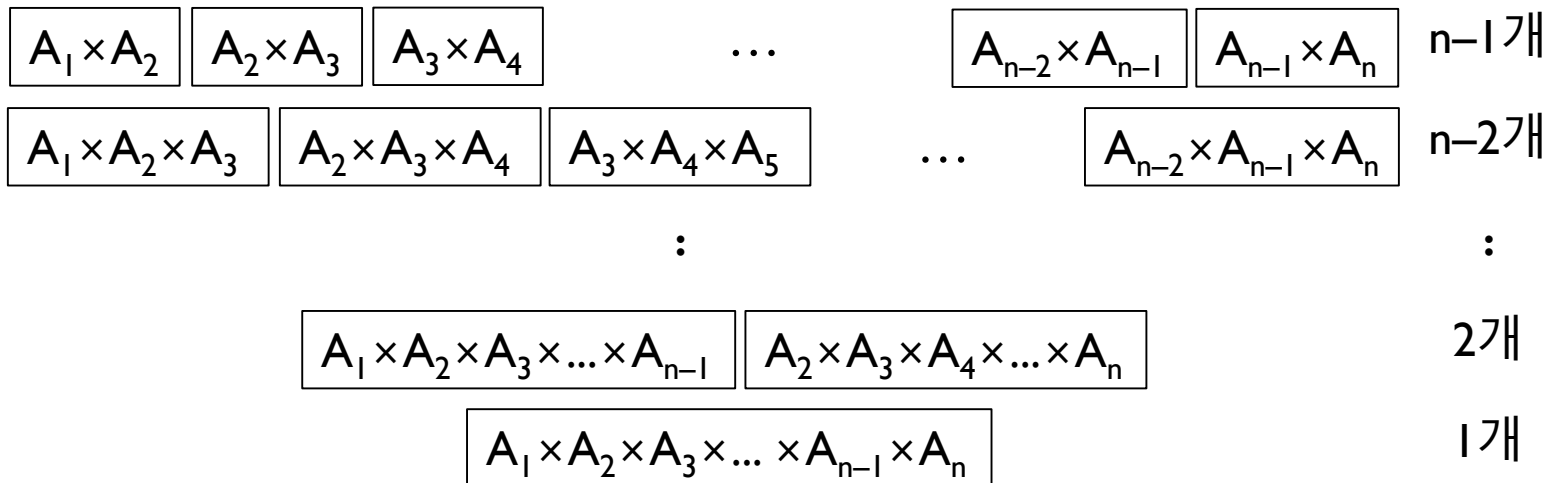
Line 5

$j=i+L$ 인데, 이는 행렬 $A_i \times \dots \times A_j$ 에 대한 원소 간의 최소 곱셈 횟수, 즉, $C[i, j]$ 를 계산하기 위한 것이다. 따라서

- $L=1$ 일 때,
 - $i=1$: $j=1+1=2$ ($A_1 \times A_2$ 를 계산하기 위하여),
 - $i=2$: $j=2+1=3$ ($A_2 \times A_3$ 을 계산하기 위하여),
 - $i=3$: $j=3+1=4$ ($A_3 \times A_4$ 를 계산하기 위하여),
 - ...
 - $i=n-L=n-1$: $j=(n-1)+1=n$ ($A_{n-1} \times A_n$ 을 계산하기 위하여)따라서 크기가 2인 부분문제의 수가 총 $(n-1)$ 개이다.
- $L=2$ 일 때,
 - $i=1$: $j=1+2=3$ ($A_1 \times A_2 \times A_3$ 을 계산하기 위하여),
 - $i=2$: $j=2+2=4$ ($A_2 \times A_3 \times A_4$ 를 계산하기 위하여),
 - $i=3$: $j=3+2=5$ ($A_3 \times A_4 \times A_5$ 를 계산하기 위하여),
 - ...
 - $i=n-L=n-2$: $j=(n-2)+2=n$ ($A_{n-2} \times A_{n-1} \times A_n$ 을 계산하기 위해)따라서 크기 3인 부분문제의 수가 총 $(n-2)$ 개이다.

연속 행렬 곱셈 알고리즘

Line 5	<ul style="list-style-type: none"> • $L=3$일 때, $A_1 \times A_2 \times A_3 \times A_4, A_2 \times A_3 \times A_4 \times A_5, \dots, A_{n-3} \times A_{n-2} \times A_{n-1} \times A_n$을 계산한다. 크기가 4인 부분문제의 수가 총 $(n-3)$개이다. ... • $L=n-2$일 때, 2개의 부분문제 $A_1 \times A_2 \times \dots \times A_{n-1}, A_2 \times A_3 \times \dots \times A_n$을 계산한다. • $L=n-1$일 때, $i=1$이면 $j=1+(n-1)=n$이고, 주어진 문제 입력인 $A_1 \times A_2 \times \dots \times A_n$을 계산한다.
--------	---



연속 행렬 곱셈 알고리즘

Line 6	최소 곱셈 횟수를 찾기 위해 $C[i, j] = \infty$ 로 초기화시킨다.
Line 7~10	for-루프는 k 가 i 부터 $(j-1)$ 까지 변하면서 어떤 부분문제를 먼저 계산하면 곱셈 횟수가 최소인지를 찾아서 최종적으로 $C[i, j]$ 에 그 값을 저장한다. 즉, k 가 $A_i \times A_{i+1} \times \dots \times A_j$ 를 2개의 부분문제로 나누어 어떤 경우에 곱셈 횟수가 최소인지를 찾는데, 여기서 부분문제간의 함축적 순서가 존재함을 알 수 있다.

$$(A_i) \times (A_{i+1} \times A_{i+2} \times A_{i+3} \times \dots \times A_j)$$

$k=i$ 일 때

$$(A_i \times A_{i+1}) \times (A_{i+2} \times A_{i+3} \times \dots \times A_j)$$

$k=i+1$ 일 때

$$(A_i \times A_{i+1} \times A_{i+2}) \times (A_{i+3} \times \dots \times A_j)$$

$k=i+2$ 일 때

:

:

$$(A_i \times A_{i+1} \times A_{i+2} \times \dots \times A_{j-1}) \times (A_j)$$

$k=j-1$ 일 때

연속 행렬 곱셈 알고리즘

Line 8

2개의 부분문제로 나뉜 각 경우에 대한 곱셈 횟수를 계산한다. 첫 번째 부분문제의 해 $C[i, k]$, 두 번째 부분문제의 해 $C[k+1, j]$ 와 $d_{i-1}d_kd_j$ 를 더한다. 여기서 $d_{i-1}d_kd_j$ 를 더하는 이유는 두 부분문제들이 각각 $d_{i-1} \times d_k$ 행렬과 $d_k \times d_j$ 행렬이고, 두 행렬을 곱하는데 필요한 원소 간의 곱셈 횟수가 $d_{i-1}d_kd_j$ 이기 때문이다. 다음은 k 값의 변화에 따른 2개의 부분문제에 해당하는 행렬을 각각 보여주고 있다.

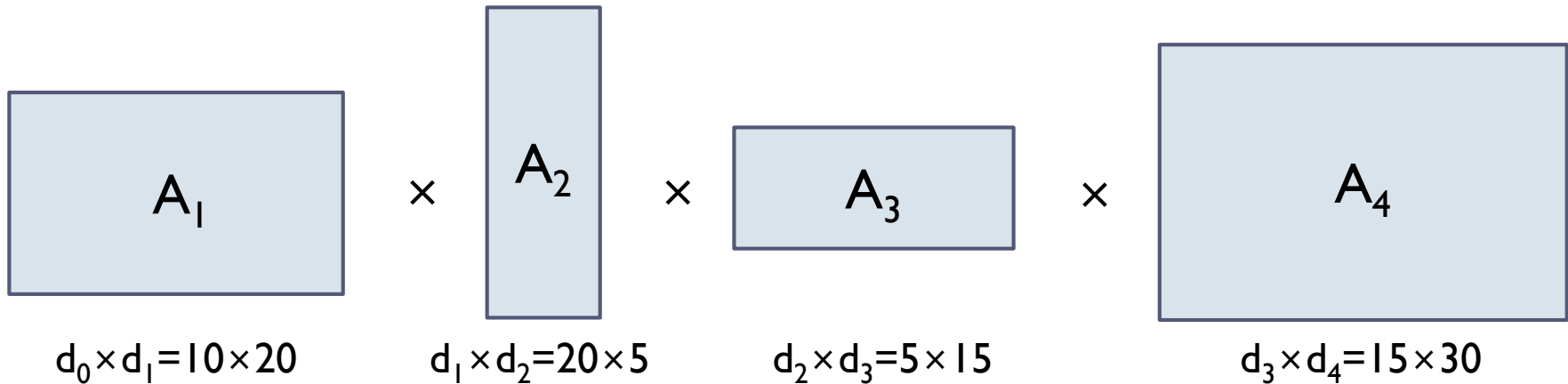
(A_i)	\times	$(A_{i+1} \times A_{i+2} \times \dots \times A_j)$	$k=i$ 일 때
$d_{i-1}d_i$		d_id_j	
$(A_i \times A_{i+1})$	\times	$(A_{i+2} \times A_{i+3} \times \dots \times A_j)$	$k=i+1$ 일 때
$d_{i-1}d_{i+1}$		$d_{i+1}d_j$	
$(A_i \times A_{i+1} \times A_{i+2})$	\times	$(A_{i+3} \times \dots \times A_j)$	$k=i+2$ 일 때
$d_{i-1}d_{i+2}$		$d_{i+2}d_j$	
	:		:
$(A_i \times A_{i+1} \times \dots \times A_{j-1})$	\times	(A_j)	$k=j-1$ 일 때
$d_{i-1}d_{j-1}$		$d_{j-1}d_j$	

연속 행렬 곱셈 알고리즘

Line 9~10	Line 8에서 계산된 곱셈 횟수가 바로 직전까지 계산되어 있는 $C[i, j]$ 보다 작으면 그 값으로 $C[i, j]$ 를 갱신하며, $k=(j-1)$ 일 때까지 수행되어 최종적으로 가장 작은 값이 $C[i, j]$ 에 저장된다.
Line 11	주어진 문제의 해가 있는 $C[1, n]$ 을 리턴한다.

MatrixChain 예제

- ▶ A_1 이 10×20 , A_2 가 20×5 , A_3 이 5×15 , A_4 가 15×30 이다.



MatrixChain 예제

- ▶ Line 1~2에서 $C[1, 1]=C[2, 2]=C[3, 3]=C[4, 4]= 0$ 으로 초기화시킨다.
- ▶ Line 3에서 L 이 1부터 $(4-1)=3$ 까지 변하고, 각각의 L 값에 대하여, i 가 변화하며 $C[i, j]$ 를 계산한다.
- ▶ $L=1$ 일 때, i 는 1부터 $(n-L)=4-1=3$ 까지 변한다.
 - ▶ $i=1$ 이면 $j=i+L=1+1=2$ 이므로, $A_1 \times A_2$ 를 위해 line 6에서 $C[1, 2]=\infty$ 로 초기화하고, line 7의 k 는 1부터 $(j-1)=2-1=1$ 까지 변하므로 사실 $k=1$ 일때 1번만 수행된다. Line 8에서 $\text{temp} = C[1, 1] + C[2, 2] + d_0d_1d_2 = 0 + 0 + (10 \times 20 \times 5) = 1000$ 이 되고, line 9에서 현재 $C[1, 2]=\infty$ 가 temp 보다 크므로, $C[1, 2]=1000$ 이 된다.

이때 k가 1

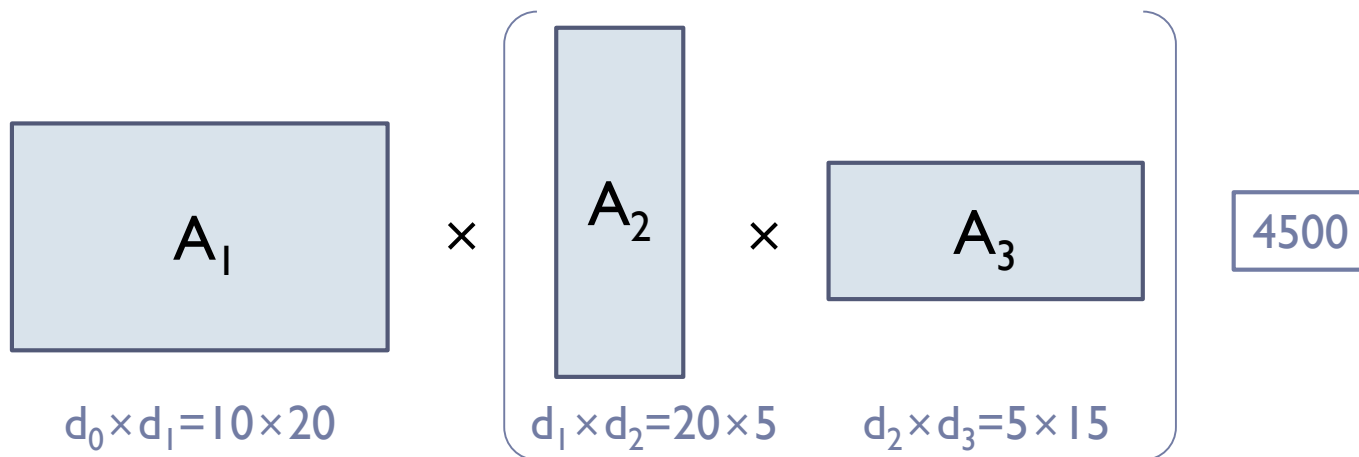
MatrixChain 예제

- ▶ $i=2$ 이면 $j=i+L=2+1=3$ 이므로, $A_2 \times A_3$ 을 위해 line 6에서 $C[2, 3] = \infty$ 로 초기화하고, line 7의 k 는 2부터 $(j-1)=3-1=2$ 까지 변하므로 $k=2$ 일 때 역시 1번만 수행된다. Line 8에서 $\text{temp} = C[2, 2] + C[3, 3] + d_1 d_2 d_3 = 0 + 0 + (20 \times 5 \times 15) = 1500$ 이 되고, line 9에서 현재 $C[2, 3] = \infty$ 가 temp 보다 크므로, $C[2, 3] = 1500$ 이 된다.
- ▶ $i=3$ 이면 $A_3 \times A_4$ 에 대해 $C[3, 4] = 2250$ 이 된다. Line 8에서 $\text{temp} = C[3, 3] + C[4, 4] + d_2 d_3 d_4 = 0 + 0 + (5 \times 15 \times 30) = 2250$ 이 되기 때문이다.

C	1	2	3	4
1	0	1000		
2		0	1500	
3			0	2250
4				0

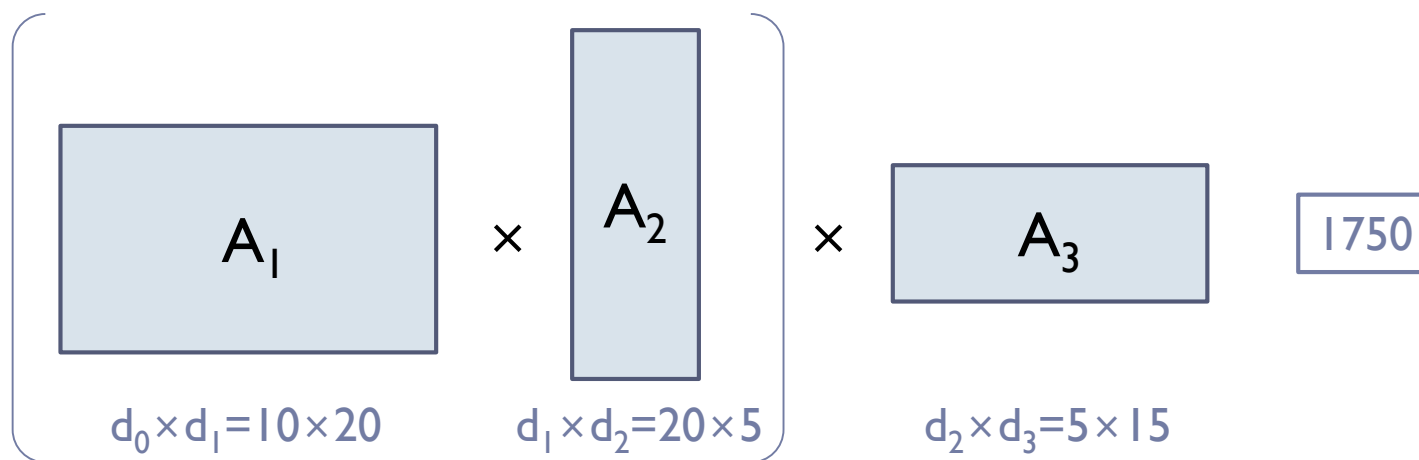
MatrixChain 예제

- ▶ $L=2$ 일 때 i 는 1부터 $(n-L)=4-2=2$ 까지 변한다.
 - ▶ $i=1$ 이면 $j=i+L=1+2=3$ 이므로, $A_1 \times A_2 \times A_3$ 을 계산하기 위해 line 6에서 $C[1, 3]=\infty$ 로 초기화하고, line 7의 k 는 1부터 $(j-i)=3-1=2$ 까지 변하므로, $k=1$ 과 $k=2$ 일 때 2번 수행된다.
 - $k=1$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[1, 1] + C[2, 3] + d_0d_1d_3 = 0 + 1500 + (10 \times 20 \times 15) = 4500$ 이 되고, line 9에서 현재 $C[1, 3]=\infty$ 이고 temp 보다 크므로, $C[1, 3]=4500$ 이 된다.



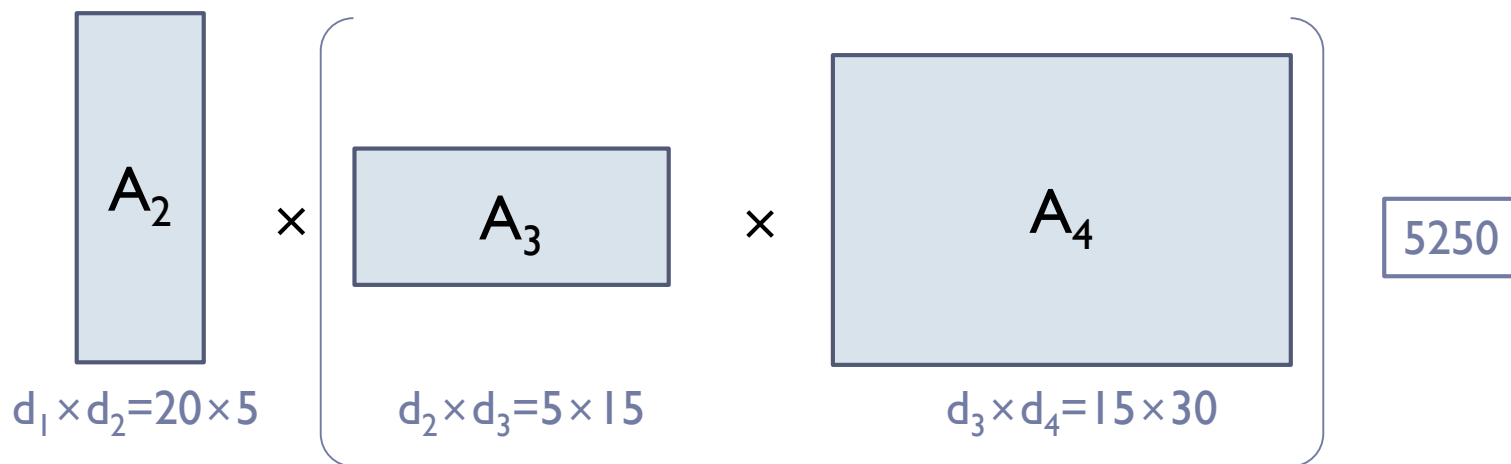
MatrixChain 예제

- $k=2$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[1, 2] + C[3, 3] + d_0d_2d_3 = 1000 + 0 + (10 \times 5 \times 15) = 1750$ 이 되고, line 9에서 현재 $C[1, 3]=4500$ 인데 temp 보다 크므로, $C[1, 3]=1750$ 으로 갱신된다.



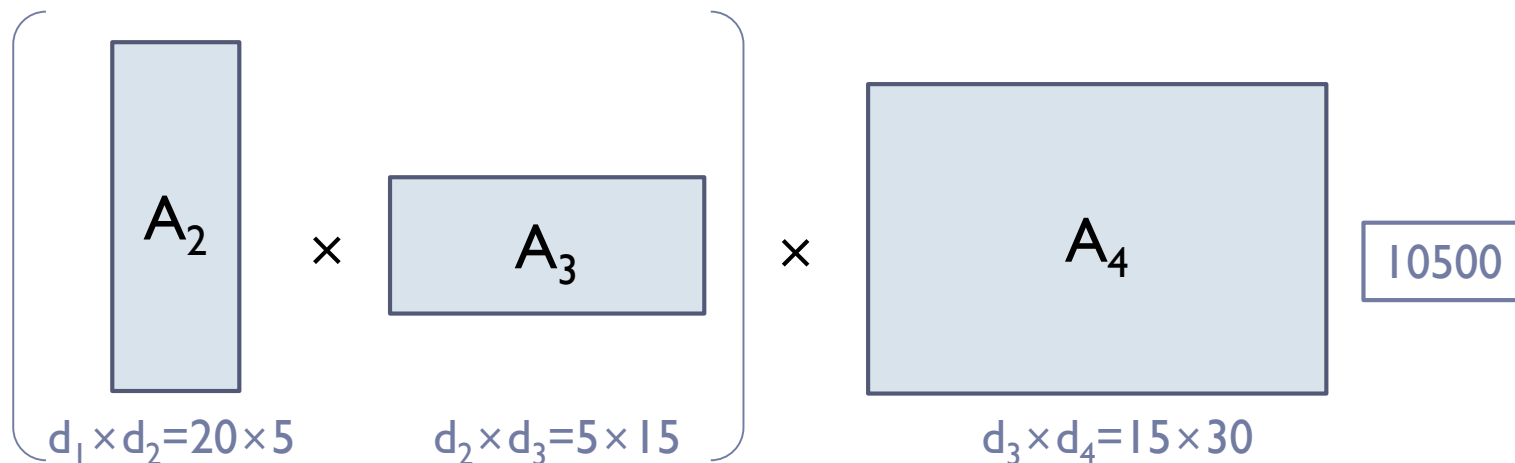
MatrixChain 예제

- ▶ $i=2$ 이면 $j=i+L=2+2=4$ 이므로, $A_2 \times A_3 \times A_4$ 를 계산하기 위해 line 6에서 $C[2, 4]=\infty$ 로 초기화하고, line 7의 k 는 2부터 $(j-1)=3$ 까지 변하므로, $k=2$ 와 $k=3$ 일 때 2번 수행된다.
 - $k=2$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[2, 2] + C[3, 4] + d_1d_2d_4 = 0 + 2250 + (20 \times 5 \times 30) = 5250$ 이 되고, line 9에서 현재 $C[2, 4]=\infty$ 가 temp 보다 크므로, $C[2, 4]=5250$ 이 된다.



MatrixChain 예제

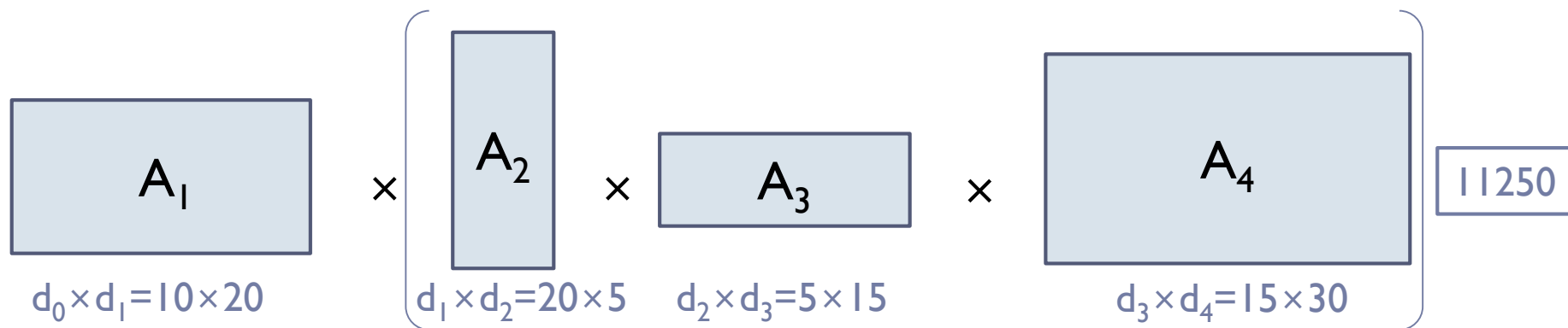
- $k=3$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[2, 3] + C[4, 4] + d_1d_3d_4 = 1500 + 0 + (20 \times 15 \times 30) = 10500$ 이 된다. 그러나 line 9에서 현재 $C[2, 4]=5250$ 이 temp 보다 작으므로, 그대로 $C[2, 4]=5250$ 이다.



C	1	2	3	4
1	0	1000	1750	
2		0	1500	5250
3			0	2250
4				0

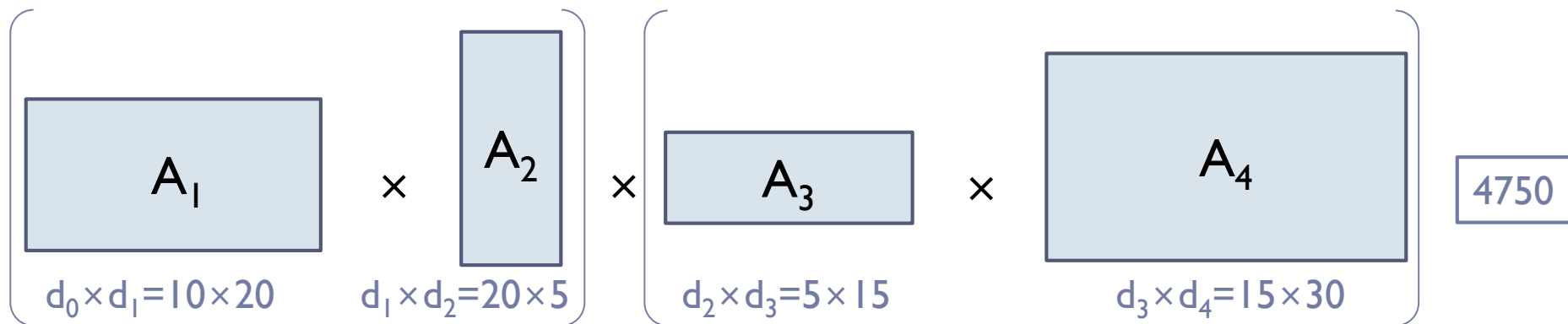
MatrixChain 예제

- ▶ $L=3$ 일 때 i 는 1부터 $(n-L)=4-3=1$ 까지 이므로 $i=1$ 일 때만 수행된다.
 - ▶ $i=1$ 이면 $j=i+L=1+3=4$ 이므로, $A_1 \times A_2 \times A_3 \times A_4$ 를 계산하기 위해 line 6에서 $C[1, 4]=\infty$ 로 초기화하고, line 7의 k 는 1부터 $(j-1)=4-1=3$ 까지 변하므로, $k=1, k=2, k=3$ 일 때 각각 수행된다.
 - $k=1$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[1, 1] + C[2, 4] + d_0d_1d_4 = 0 + 5250 + (10 \times 20 \times 30) = 11250$ 이 되고, line 9에서 현재 $C[1, 4]=\infty$ 가 temp 보다 크므로, $C[1, 4]=11250$ 이 된다.

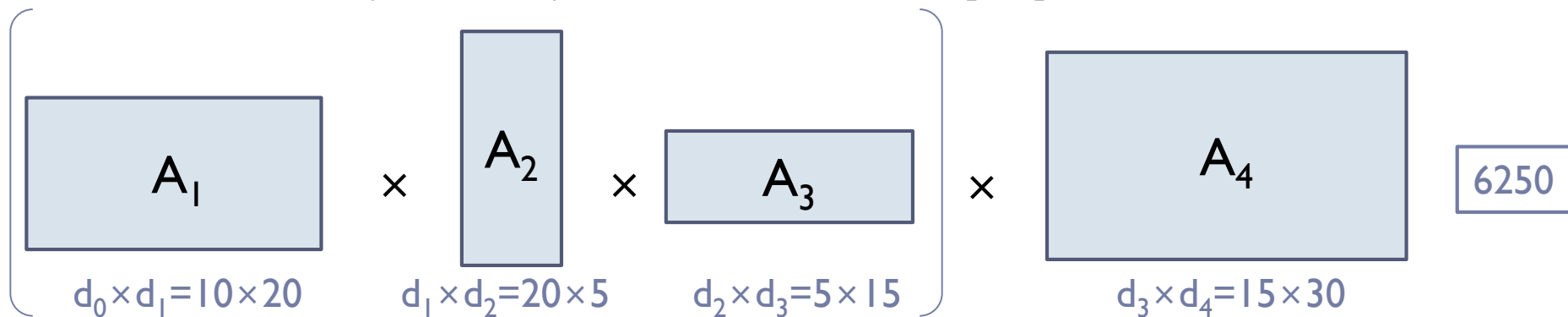


MatrixChain 예제

- $k=2$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[1, 2] + C[3, 4] + d_0d_2d_4 = 1000 + 2250 + (10 \times 5 \times 30) = 4750$ 이 되고, line 9에서 현재 $C[1, 4] = 11250$ 이 temp 보다 크므로, $C[1, 4]=4750$ 으로 갱신된다.



- $k=3$ 일 때, line 8에서 $\text{temp} = C[i, k] + C[k+1, j] + d_{i-1}d_kd_j = C[1, 3] + C[4, 4] + d_0d_3d_4 = 1750 + 0 + (10 \times 15 \times 30) = 6250$ 이 되고, 현재 $C[1, 4] = 4750$ 이 그대로 유지된다.



MatrixChain 예제

- ▶ 따라서 이 예제의 최종해는 4750번이다.
- ▶ 먼저 $A_1 \times A_2$ 를 계산하고, 그 다음엔 $A_3 \times A_4$ 를 계산하여, 각각의 결과를 곱하는 것이 가장 효율적이다. 다음은 알고리즘이 수행된 후의 배열 C이다.

C	1	2	3	4
1	0	1000	1750	4750
2		0	1500	3250
3			0	2250
4				0

3
2
1

시간복잡도

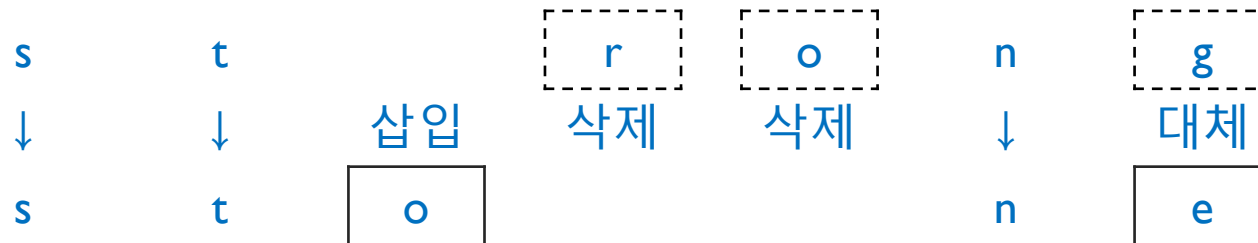
- ▶ MatrixChain 알고리즘의 시간복잡도는 2차원 배열 C 만 보더라도 쉽게 알 수 있다.
- ▶ 배열 C 가 $n \times n$ 이고, 원소의 수는 n^2 인데, 거의 $1/2$ 정도의 원소들의 값을 계산해야 한다.
- ▶ 그런데 하나의 원소, 즉 $C[i, j]$ 를 계산하기 위해서는 k -루프가 최대 $(n-1)$ 번 수행되어야 한다. 과호 계산양식
- ▶ 따라서 MatrixChain 알고리즘의 시간복잡도는 $O(n^2) \times O(n)$
= $O(n^3)$ 이다.

5.3 편집 거리 문제

- ▶ 문서 편집기를 사용하는 중에 하나의 스트링(문자열) S 를 수정하여 다른 스트링 T 로 변환시키고자 할 때, 삽입(insert), 삭제(delete), 대체(substitute) 연산이 사용된다.
- ▶ S 를 T 로 변환시키는데 필요한 최소의 편집 연산 횟수를 편집 거리(edit distance)라고 한다.
- ▶ 편집 거리 문제는 편집 거리를 찾는 문제이다.

편집 거리 문제

- ▶ 예를 들어, 'strong'을 'stone'으로 편집하여 보자.



- ▶ 위의 편집에서는 's'와 't'는 그대로 사용하고, 'o'를 삽입하고, 'r'과 'o'를 삭제한다.
- ▶ 그 다음에는 'n'을 그대로 사용하고, 마지막으로 'g'를 'e'로 대체시켜, 총 4회의 편집 연산이 수행되었다.

편집 거리 문제

- ▶ 반면에 아래의 편집에서는 ‘s’와 ‘t’는 그대로 사용한 후, ‘r’을 삭제하고, ‘o’와 ‘n’을 그대로 사용한 후, ‘g’를 ‘e’로 대체시켜, 총 2회의 편집 연산만이 수행되었고, 이는 최소 편집 횟수이다.

s	t	r	o	n	g
↓	↓	삭제	↓	↓	대체
s	t		o	n	e

- ▶ 이처럼 S를 T로 바꾸는데 어떤 연산을 어느 문자에 수행하는가에 따라서 편집 연산 횟수가 달라진다.

편집 거리 문제

- ▶ 편집 거리 문제를 동적 계획 알고리즘으로 해결하려면 부분문제들을 어떻게 표현해야 할까?
- ▶ ‘strong’을 ‘stone’으로 편집하려는데, 만일 각 접두부(prefix)에 대해서, 예를 들어 ‘stro’를 ‘sto’로 편집할 때의 편집 거리를 미리 알고 있으면, 각 스트링의 나머지 부분에 대해서, 즉 ‘ng’를 ‘ne’로의 편집에 대해서 편집 거리를 찾음으로써 주어진 입력에 대한 편집 거리를 구할 수 있다.

S =

s	t	r	o
---	---	---	---

 n g

T =

s	t	o
---	---	---

 n e

1 2 3 4

1 2 3

편집 거리 문제

- ▶ 부분문제를 정의하기 위해서 스트링 S 와 T 의 길이가 각각 m 과 n 이라 하고, S 와 T 의 각 문자를 다음과 같이 s_i 와 t_j 라고 하자. 단, $i = 1, 2, \dots, m$ 이고, $j = 1, 2, \dots, n$ 이다.

$S = s_1 \quad s_2 \quad s_3 \quad s_4 \quad \dots \quad s_m$

$T = t_1 \quad t_2 \quad t_3 \quad t_4 \quad \dots \quad t_n$

- ▶ 부분문제의 정의: $E[i, j]$ 는 S 의 접두부 i 개 문자를 T 의 접두부 j 개 문자로 변환시키는데 필요한 최소 편집 연산 횟수, 즉 편집 거리이다.
 - ▶ 예를 들어, 'strong'을 'stone'에 대해서, 'stro'를 'sto'로 바꾸기 위한 편집 거리를 찾는 문제는 $E[4, 3]$ 이 되고, 점진적으로 $E[6, 5]$ 를 해결하면 문제의 해를 찾게 된다.

편집 거리 문제

- ▶ 다음 예제에서 처음 몇 개의 부분문제를 계산하여 보자.

	1	2	3	4	5	6
S	s	t	r	o	n	g
T	s	t	o	n	e	

- ▶ $s_1 \rightarrow t_1$ ['s'를 's'로 편집]: $E[1, 1]=0$. 왜냐하면 $s_1=t_1='s'$ 이기 때문이다.
- ▶ $s_1 \rightarrow t_1 t_2$ ['s'를 'st'로 편집]: $E[1, 2]=1$ 이다. 왜냐하면 $s_1=t_1='s'$ 이고, 't'를 삽입하는데 1회의 연산이 필요하기 때문이다.
- ▶ $s_1 s_2 \rightarrow t_1$ ['st'를 's'로 편집]: $E[2, 1]=1$ 이다. 왜냐하면 $s_1=t_1='s'$ 이고, 't'를 삭제하는데 1회의 연산이 필요하기 때문이다.
- ▶ $s_1 s_2 \rightarrow t_1 t_2$ ['st'를 'st'로 편집]: $E[2, 2]=0$ 이다. 왜냐하면 $s_1=t_1='s'$ 이고, $s_2=t_2='t'$ 이기 때문이다.

편집 거리 문제

- ▶ 부분문제 $s_1s_2s_3s_4 \rightarrow t_1t_2t_3$ ['stro'를 'sto'로 편집], 즉 $E[4, 3]$ 은 어떻게 계산하여야 할까?
 - ▶ 부분문제 $s_1s_2s_3s_4 \rightarrow t_1t_2$ ['stro'를 'st'로 편집], 즉 $E[4, 2]$ 의 해를 알면, $t_3='o'$ 를 삽입하면 된다. 따라서 이때의 편집 연산 횟수는 $E[4, 2]+1$ 이다.
 - ▶ 부분문제 $s_1s_2s_3 \rightarrow t_1t_2t_3$ ['str'을 'sto'로 편집], 즉 $E[3, 3]$ 의 해를 알면, $s_4='o'$ 를 삭제하면 된다. 따라서 이때의 편집 연산 횟수는 $E[3, 3]+1$ 이다.
 - ▶ 부분문제 $s_1s_2s_3 \rightarrow t_1t_2$ ['str'을 'st'로 편집], 즉 $E[3, 2]$ 의 해를 알면, $s_4='o'$ 를 $t_3='o'$ 로 편집하는데 필요한 연산을 계산하면 된다. 이 경우에는 2개의 문자가 같으므로 편집할 필요가 없다. 따라서 이때의 편집 연산 횟수는 $E[3, 2]$ 이다.

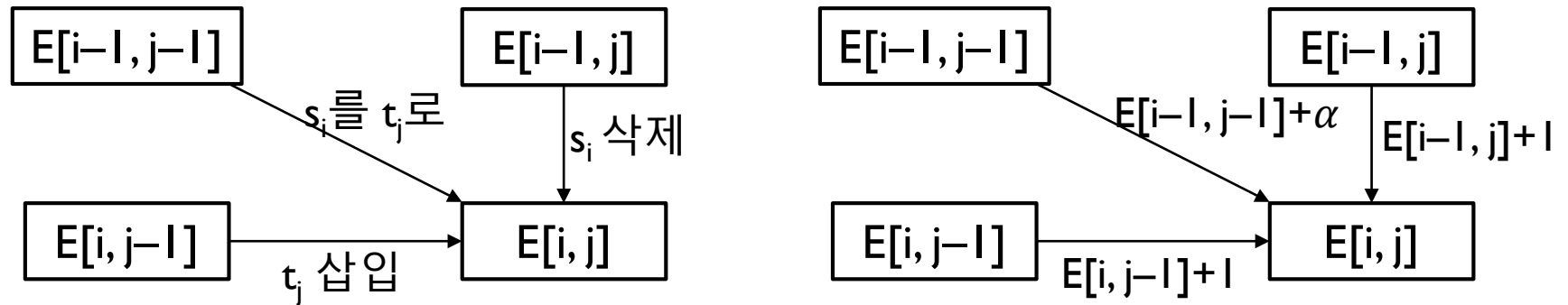
편집 거리 문제

- ▶ 따라서 $E[4, 3]$ 의 편집 거리는 3가지 부분문제들의 해, 즉, $E[4, 2]$, $E[3, 3]$, $E[3, 2]$ 의 편집 거리를 알면 된다.
- ▶ $E[4, 2]=2$, $E[3, 3]=1$, $E[3, 2]=1$ 이므로, $(2+1)$, $(1+1)$, 1 중에서 최소값인 1 이 $E[4, 3]$ 의 편집 거리가 된다.

E	T	ε	s	t	o
S	i / j	0	1	2	3
ε	0	0	1	2	3
s	1	1	0	1	2
t	2	2	1	0	1
r	3	3	2	1	1
o	4	4	3	2	1

편집 거리 문제

- ▶ 일반적으로 $E[i-1, j]$, $E[i, j-1]$, $E[i-1, j-1]$ 의 해가 미리 계산되어 있으면 $E[i, j]$ 를 계산할 수 있다. 그러므로 편집 거리 문제의 부분문제간의 함축적인 순서는 다음과 같다.



- ▶ $E[i, j]$ 의 왼쪽에 있는 $E[i, j-1]$ 는 $s_1 s_2 \dots s_i$ 와 $t_1 t_2 \dots t_{j-1}$ 까지의 해이므로 t_j 를 삽입한다면, $(E[i, j-1] + 1)$ 이 $s_1 s_2 \dots s_i$ 를 $t_1 t_2 \dots t_j$ 로 만드는데 필요한 연산 횟수가 된다.

편집 거리 문제

- ▶ $E[i, j]$ 의 위쪽에 있는 $E[i-1, j]$ 의 경우에는 $s_1 s_2 \dots s_{i-1}$ 와 $t_1 t_2 \dots t_j$ 까지의 해가 s_i 를 삭제한다면, $(E[i-1, j]+1)$ 이 $s_1 s_2 \dots s_i$ 를 $t_1 t_2 \dots t_j$ 로 만드는데 필요한 연산 횟수가 된다.
- ▶ 대각선 방향의 경우에는 연산 횟수가 $(E[i-1, j-1]+\alpha)$ 인데, 여기서 $s_i=t_j$ 이면 $\alpha=0$ 이고, $s_i \neq t_j$ 이면 $\alpha=1$ 이다. 왜냐하면 s_i 와 t_j 가 같으면 어떤 편집 연산이 필요 없고, 다르면 s_i 를 t_j 로 대체하는 연산이 필요하기 때문이다.
- ▶ 그러므로 3가지 경우 중에서 가장 작은 값을 $E[i, j]$ 의 해로서 선택한다.

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

단, $\alpha=1$ if $s_i \neq t_j$, else $\alpha=0$

편집 거리 문제

- ▶ 앞의 식을 위해서 $E[0, 0], E[1, 0], E[2, 0], \dots, E[m, 0]$ 과 $E[0, 1], E[0, 2], \dots, E[0, n]$ 이 다음과 같이 초기화되어야 한다.

	T	ε	t_1	t_2	t_3	..	t_n
S		0	1	2	3	..	n
ε	0	0	1	2	3	..	n
s_1	1	1					
s_2	2	2					
s_3	3	3					
.	.	.					
.	.	.					
s_m	m	m					

편집 거리 문제

- ▶ 2차원 배열 E 의 0번 행이 $0, 1, 2, \dots, n$ 으로 초기화된 의미는 S 의 첫 문자를 처리하기 전에, 즉 S 가 ε (공 문자열)인 상태에서 T 의 문자를 좌에서 우로 하나씩 만들어 가는데 필요한 삽입 연산 횟수를 각각 나타낸 것이다.
- ▶ $E[0, 0]=0$: T 의 첫 문자를 만들기 이전이므로, 아무런 연산이 필요 없다.
- ▶ $E[0, 1]=1$: T 의 첫 문자를 만들기 위해 ' t_1 '을 삽입해야 한다.
- ▶ $E[0, 2]=2$: T 의 처음 두 문자를 만들기 위해 ' $t_1 t_2$ '를 각각 삽입해야 한다.
- ...
- ▶ $E[0, n]=n$: T 를 만들기 위해 ' $t_1 t_2 t_3 \dots t_n$ '을 각각 삽입해야 한다.

편집 거리 문제

- ▶ 배열 E 의 0번 열이 $0, 1, 2, \dots, m$ 으로 초기화된 의미는 스트링 T 를 ε 로 만들기 위해서, S 의 문자를 위에서 아래로 하나씩 없애는데 필요한 삭제 연산 횟수를 각각 나타낸 것이다.
- ▶ $E[0, 0]=0$: S 의 첫 문자를 지우기 이전이므로, 아무런 연산이 필요 없다.
- ▶ $E[1, 0]=1$: S 의 첫 문자 ' s_1 '을 삭제해야 T 가 ε 가 된다.
- ▶ $E[2, 0]=2$: S 의 처음 두 문자 ' s_1s_2 '를 삭제해야 T 가 ε 가 된다.
- ...
- ▶ $E[m, 0]=m$: T 의 모든 문자를 삭제해야 T 가 ε 가 된다.

편집 거리 문제 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]+α}
6  return E[m, n]
```

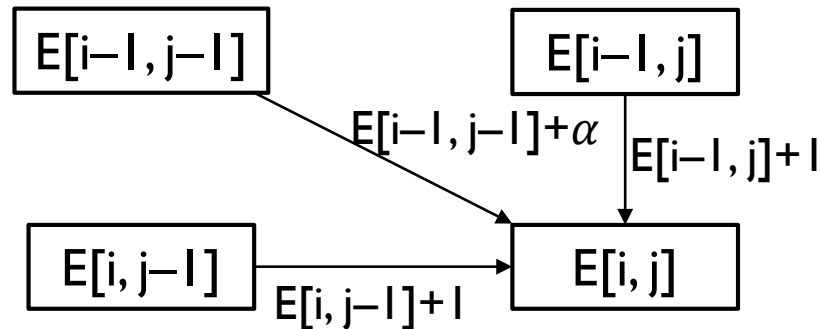
편집 거리 문제 알고리즘

Line 1~2

배열의 0번 열과 0번 행을 각각 초기화시킨다.

Line 3~5

배열을 1번 행, 2번 행, ... 순으로 채워나간다. 다음 그림과 같이 $E[i, j]$ 의 (왼쪽 원소의 값+1), (위쪽 원소의 값+1), (대각선 위쪽의 원소의 값+ α) 중에서 가장 작은 값이 $E[i, j]$ 에 저장된다.



EditDistance 예제

- ▶ 다음은 EditDistance 알고리즘이 'strong'을 'stone'으로 바꾸는데 필요한 편집 거리를 계산한 결과인 배열 E이다.

E	T	ϵ	s	t	o	n	e
S		0	1	2	3	4	5
ϵ	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
o	4	4	3	2	1	2	3
n	5	5	4	3	2	1	2
g	6	6	5	4	3	2	2

EditDistance 예제

- ▶ 배열에서 파란색 음영으로 표시된 원소가 계산되는 과정을 각각 상세히 살펴보자.
- ▶ $E[1, 1] = \min\{E[1, 0] + 1, E[0, 1] + 1, E[0, 0] + \alpha\}$
 $= \min\{(1+1), (1+1), (0+0)\} = 0$
- ▶ ‘ $E[1, 0] + 1 = 2$ ’는 S의 첫 문자를 삭제하여 $E[1, 0] = 1$ 이 되어있는 상태에서, ‘+1’은 T의 첫 문자인 $t_1 = 's'$ 를 삽입한다는 의미이다. 즉, T가 ϵ 인 상태이므로 T의 첫 문자를 삽입해야 ‘s’가 만들어진다는 뜻이다.

	T	ϵ	s
S	$\begin{matrix} i & j \\ \backslash & / \end{matrix}$	0	1
ϵ	0	0	1
s	1	1	$\Rightarrow (2)$

EditDistance 예제

- ▶ ' $E[0, 1] + 1 = 2$ '는 T의 첫 문자인 s_1 을 삽입하여 $E[0, 1] = 1$ 이 되어있는 상태에서, '+1'은 S의 첫 문자인 $s_1 = 's'$ 를 삭제한다는 의미이다. 즉, 이미 T의 첫 문자 's'가 만들어져 있는 상태이므로 S의 첫 문자를 삭제한다는 뜻이다.

	T	ϵ	(s)
S	i / j	0	1
ϵ	0	0	1
(s)	1	1	(2)

EditDistance 예제

- ▶ $E[0, 0] + \alpha = 0 + 0 = 0$ 인데 α 가 S의 첫 문자와 T의 첫 문자가 같기 때문이다. 즉, S의 1번째 문자와 T의 1번째 문자가 같으므로 아무런 연산이 필요 없이 T의 첫 문자인 's'를 만들 수 있다.
- ▶ 따라서 위의 3가지 경우의 값 중에서 최소값인 0이 $E[1, 1]$ 이 된다.

	T	ϵ	s
S	i / j	0	1
ϵ	0	0	1
s	1	1	0

EditDistance 예제

- ▶ $E[2, 2] = \min\{E[2, 1]+1, E[1, 2]+1, E[1, 1]+\alpha\} = \min\{(1+1), (1+1), (0+0)\} = 0$. 즉, 현재 T의 첫 문자 's'가 만들어져 있는 상태에서 S의 2번째 문자와 T의 2번째 문자가 같으므로 아무런 연산이 필요 없이 'st'가 만들어지는 것이다.

	T	ϵ	s	t
S	i \ j	0	1	2
ϵ	0	0	1	2
s	1	1	0	1
t	2	2	1	0

EditDistance 예제

- ▶ $E[3, 2] = \min\{E[3, 1]+1, E[2, 2]+1, E[2, 1]+\alpha\} = \min\{2+1, 0+1, 1+1\} = 1$.
즉, 이미 T의 처음 2문자 'st'가 만들어져 있으므로 S의 3번째 문자인 'r'을 삭제한다는 의미이다.

	T	ϵ	s	t
S	i \ j	0	1	2
ϵ	0	0	1	2
s	1	1	0	1
t	2	2	1	0
r	3	3	2	1

EditDistance 예제

- ▶ $E[4, 3] = \min\{E[4, 2]+1, E[3, 3]+1, E[3, 2]+\alpha\} = \min\{(2+1), (1+1), (1+0)\} = 1$. 즉, 현재 T의 처음 2문자 'st'가 만들어져 있는 상태에서 S의 4번째 문자와 T의 3번째 문자가 같으므로 아무런 연산이 필요 없이 'sto'가 만들어지는 것이다.

	T	ϵ	s	t	o
S	i \ j	0	1	2	3
ϵ	0	0	1	2	3
s	1	1	0	1	2
t	2	2	1	0	1
r	3	3	2	1	1
o	4	4	3	2	1

EditDistance 예제

- ▶ $E[5, 4] = \min\{E[5, 3]+1, E[4, 4]+1, E[4, 3]+\alpha\} = \min\{(2+1), (2+1), (1+0)\} = 1$. 즉, 현재 T의 처음 3문자 'sto'가 만들어져 있는 상태에서 S의 5번째 문자와 T의 4번째 문자가 같으므로 아무런 연산이 필요 없이 'ston'이 만들어지는 것이다.

	T	ϵ	s	t	o	n
S	i \ j	0	1	2	3	4
ϵ	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
o	4	4	3	2	1	2
n	5	5	4	3	2	1

EditDistance 예제

- ▶ $E[6, 5] = \min\{E[6, 4] + 1, E[5, 5] + 1, E[5, 4] + \alpha\} = \min\{(2+1), (2+1), (1+1)\} = 2$. 즉, 현재 T의 처음 4문자 'ston'이 만들어져 있는 상태에서 S의 6번째 문자인 'g'를 T의 5번째 문자인 'e'로 대체하여 T를 완성시킨다는 의미이다.

		T	ϵ	s	t	o	n	e
S	i \ j	0	1	2	3	4	5	
ϵ	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	
o	4	4	3	2	1	2	3	
n	5	5	4	3	2	1	2	
g	6	6	5	4	3	2	2	2

시간복잡도

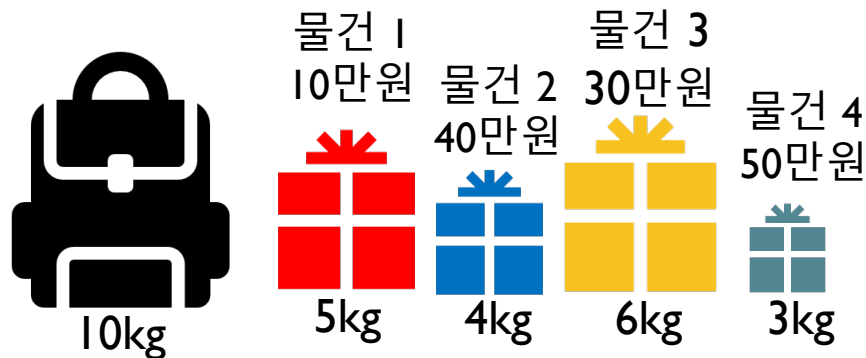
- ▶ EditDistance 알고리즘의 시간복잡도는 $O(mn)$ 이다. 단, m 과 n 은 두 스트링의 각각의 길이이다.
- ▶ 총 부분문제의 수가 배열 E 의 원소 수인 $m \times n$ 이고, 각 부분문제(원소)를 계산하기 위해서 주위의 3개의 부분문제들의 해(원소)를 참조한 후 최소값을 찾는 것이므로 $O(1)$ 시간이 걸리기 때문이다.

응용

- ▶ 2개의 스트링 사이의 편집 거리가 작으면, 이 스트링들이 서로 유사하다고 판단할 수 있다.
- ▶ 생물 정보 공학(Bioinformatics) 및 의학 분야에서 두 개의 유전자가 얼마나 유사한가를 측정하는데 활용
 - ▶ 예를 들어, 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 활용된다.
- ▶ 철자 오류 검색(Spell Checker)
- ▶ 광학 문자 인식(Optical Character Recognition)에서의 보정 시스템(Correction System)
- ▶ 자연어 번역(Natural Language Translation) 소프트웨어

5.4 배낭 문제

- ▶ 배낭(Knapsack) 문제는 n 개의 물건과 각 물건 i 의 무게 w_i 와 가치 v_i 가 주어지고, 배낭의 용량은 C 일 때, 배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제이다.
- ▶ 배낭에 담은 물건의 무게의 합이 C 를 초과하지 말아야 하고, 각 물건은 1개씩만 있다.
- ▶ 이러한 배낭 문제를 **0-1 배낭 문제**라고 하는데, 이는 각 물건이 배낭에 담기지 않는 경우는 '0', 담긴 경우는 '1'로 여기기 때문이다.



배낭 문제

- ▶ 배낭 문제는 제한적인 입력에 대해서 동적 계획 알고리즘으로 해결할 수 있다.
 - ▶ 먼저 배낭 문제의 부분문제를 찾아내기 위해 문제의 주어진 조건을 살펴보면 물건, 물건의 무게, 물건의 가치, 배낭의 용량, 모두 4가지의 요소가 있다.
 - ▶ 이중에서 물건과 물건의 무게는 부분문제를 정의하는데 반드시 필요하다.
 - ▶ 왜냐하면 배낭이 비어 있는 상태에서 시작하여 물건을 하나씩 배낭에 담는 것과 안 담는 것을 현재 배낭에 들어 있는 물건의 가치의 합에 근거하여 결정해야 하기 때문이다.
 - ▶ 또한 물건을 배낭에 담으려고 할 경우에 배낭 용량의 초과 여부를 검사해야 한다.

배낭 문제의 부분문제

- ▶ 배낭 문제의 부분문제 정의를 위해 물건은 하나씩 차례로 고려하면 되지만, 물건의 무게는 각각 다를 수 있기 때문에, 무게에 대해서는 배낭의 용량이 $0(\text{kg})$ 으로부터 $1(\text{kg})$ 씩 증가하여 입력으로 주어진 용량인 $C(\text{kg})$ 이 될 때까지 변화시켜 가며 물건을 배낭에 담는 것이 가치가 더 커지는지를 결정해야 한다.
- ▶ 원래의 배낭 용량은 $C(\text{kg})$ 이지만, 배낭 용량이 $0(\text{kg})$ 부터 $1(\text{kg})$ 씩 증가할 경우의 용량을 ‘임시’ 배낭 용량 이라고 부르자.
- ▶ 배낭 문제의 부분문제를 아래와 같이 정의할 수 있다.
 $K[i, w]$ = 물건 $1 \sim i$ 까지만 고려하고, 임시 배낭 용량이 w 일 때의 최대 가치. 단, $i = 1, 2, \dots, n$ 이고, $w = 1, 2, 3, \dots, C$ 이다.
- ▶ 문제의 최적해는 $K[n, C]$ 이다.

배낭 문제 알고리즘

Knapsack

입력: 배낭의 용량 C , n 개의 물건과 각 물건 i 의 무게 w_i 와 가치 v_i , 단, $i=1, 2, \dots, n$
 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 배낭에 담기 위해
                                   고려하지 않았을 때
3  for i = 1 to n {
4      for w = 1 to C {           // w는 배낭의 임시 용량이고, 마지막에는 w=C가
                                   되어서 배낭의 용량이 된다.
5          if ( w_i > w )         // 물건 i의 무게가 임시 배낭 용량을 초과하면
6              K[i, w] = K[i-1, w]  // 넣으려는 물건의 무게가 배낭에 들어가지 않음
7          else                   // 물건 i를 배낭에 담지 않을 경우와 담을 경우 고려
8              K[i, w] = max{ K[i-1, w], K[i-1, w-w_i]+v_i }
          }
9  return K[n, C]
    
```

Handwritten notes:

- 1~n (next to line 3)
- 1~C (next to line 4)
- 이전까지 넣었던 것 (next to line 6)
- 이번꺼를 넣겠다. (next to line 8)
- 이전까지 넣었던 것 안하고 나머지 물건이 값이 더 있는지 (next to line 9)
- i의 무게를 빼고 (next to line 8)

배낭 문제 알고리즘

Line 1	2차원 배열 K 의 0번 열을 0으로 초기화시킨다. 그 의미는 배낭의 <u>임시 용량이 0</u> 일 때, 물건 1~ n 까지 각각 배낭에 담아보려고 해도 배낭에 담을 수 없으므로 그에 대한 각각의 가치는 0일 수밖에 없다는 뜻이다.
Line 2	0번 행의 각 원소를 0으로 초기화시킨다. 여기서 물건 0이란 어떤 물건도 배낭에 담으려고 고려하지 않는다는 뜻이다. 따라서 배낭의 용량을 0에서 C 까지 각각 증가시켜도 담을 물건이 없으므로 각각의 최대 가치는 0이다.
Line 3~8	물건을 1에서 n 까지 하나씩 고려하여 배낭의 임시 용량을 1에서 C 까지 각각 증가시키며, 다음을 수행한다.

배낭 문제 알고리즘

Line 5~6	현재 배낭에 담아보려고 고려하는 물건 i 의 무게 w_i 가 임시 배낭 용량 w 보다 크면 물건 i 를 배낭에 담을 수 없 으므로, 물건 i 까지 고려했을 때의 최대 가치 $K[i, w]$ 는 물 건 $(i-1)$ 까지 고려했을 때의 최대 가치 <u>$K[i-1, w]$가 된다.</u> //
Line 7~8 <i>else</i>	만일 현재 고려하는 물건 i 의 무게 w_i 가 현재 배낭의 용 량 w 보다 같거나 작으면, 물건 i 를 배낭에 담을 수 있다. 그러나 현재 상태에서 물건 i 를 추가로 배낭에 담으면 배낭의 무게가 <u>$(w+w_i)$</u> 로 늘어난다. 따라서 현재의 배낭 용량인 w 를 초과하게 되어, 물건 i 를 추가로 담을 수는 없다.

남아있는 $K[i-1, w]$ 가 된다.

배낭 문제 알고리즘

물건 1~(i-1)까지 고려하여
현재 배낭의 용량이 w 인
경우의 최대 가치

배낭에서 물건 i 를
담을 공간을 마련

$K[i-1, w-w_i]$

+

물건 i 의 가치 v_i

$K[i-1, w]$

0
물건 i
1

$K[i, w]$

물건 1~(i-1)까지 고려하여
현재 배낭의 용량이 $(w-w_i)$
인 경우의 최대 가치

배낭 문제 알고리즘

Line 7~8

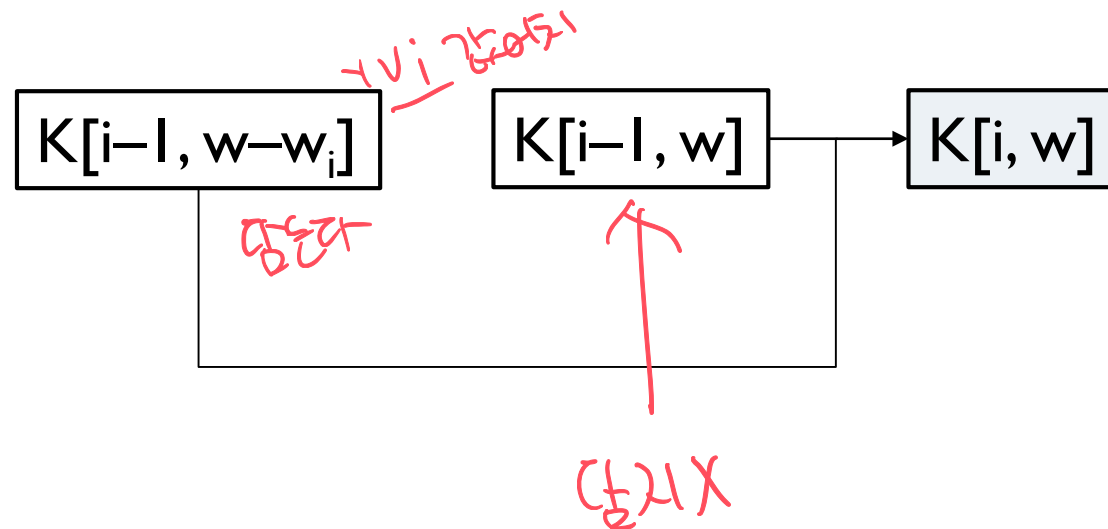
그러므로 앞의 그림에서와 같이, 물건 i 를 배낭에 담기 위해서는 2가지 경우를 살펴보아야 한다.

- 1) 물건 i 를 배낭에 담지 않는 경우, $K[i, w] = K[i-1, w]$ 가 된다.
- 2) 물건 i 를 배낭에 담는 경우, 현재 무게인 w 에서 물건 i 의 무게인 w_i 를 뺀 상태에서 물건을 $(i-1)$ 까지 고려했을 때의 최대 가치인 $K[i-1, w-w_i]$ 와 물건 i 의 가치 v_i 의 합이 $K[i, w]$ 가 되는 것이다.

Line 8에서는 이 2가지 경우 중에서 큰 값이 $K[i, w]$ 가 된다.

배낭 문제 알고리즘

- ▶ 배낭 문제의 부분문제간의 함축적 순서는 다음과 같다. 즉, 2개의 부분문제 $K[i-1, w-w_i]$ 과 $K[i-1, w]$ 가 미리 계산되어 있어야만 $K[i, w]$ 를 계산할 수 있다.



Knapsack 예제

- ▶ 배낭의 용량 $C=10\text{kg}$ 이고, 각 물건의 무게와 가치는 다음과 같다.



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

Knapsack 예제

- ▶ Line 1~2: 아래와 같이 배열의 0번 행과 0번 열의 각 원소를 0으로 초기화한다.

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										

Knapsack 예제

- ▶ Line 3~8: line 3에서는 물건을 하나씩 고려하기 위해서 물건 번호 i 가 1부터 4까지 변하며, line 4에서는 배낭의 임시 용량 w 가 1kg씩 증가되어 마지막에는 배낭의 용량인 10kg이 된다.
- ▶ $i=1$ 일 때(즉, 물건 1만을 고려한다.)
 - ▶ $w=1$ (배낭의 용량이 1kg)일 때, 물건 1을 배낭에 담아보려고 한다. 그러나 $w_1 > w$ 이므로, (즉, 물건 1의 무게가 5kg이므로, 배낭에 담을 수 없기 때문에) $K[1, 1] = K[i-1, w] = K[1-1, 1] = K[0, 1] = 0$ 이다. 즉, $K[1, 1]=0$ 이다.
 - ▶ $w=2, 3, 4$ 일 때, 각각 $w_1 > w$ 이므로, 물건 1을 담을 수 없다. 따라서 각각 $K[1, 2]=0, K[1, 3]=0, K[1, 4]=0$ 이다. 즉, 배낭의 용량을 4kg까지 늘려도 5kg의 물건 1을 배낭에 담을 수 없다.

첫번째 물건 1개.

Knapsack 예제

- ▶ $w=5$ (배낭의 용량이 5kg)일 때, 물건 1을 배낭에 담을 수 있다. 왜냐하면 $w_1=w$ 이므로, 즉 물건 1의 무게가 5kg이기 때문이다.

$$\begin{aligned}K[1, 5] &= \max\{K[i-1, w], K[i-1, w-w_i]+v_i\} \\&= \max\{K[1-1, 5], K[1-1, 5-5]+10\} \\&= \max\{K[0, 5], K[0, 0]+10\} \\&= \max\{0, 0+10\} \\&= \max\{0, 10\} = 10\end{aligned}$$

- ▶ $w=6, 7, 8, 9, 10$ 일 때, 각각의 경우가 $w=5$ 일 때와 동일하게 물건 1을 담을 수 있다. 따라서 각각 $K[1, 6] = K[1, 7] = K[1, 8] = K[1, 9] = K[1, 10] = 10$ 이다.

$9-1=0$

안담는다

Knapsack 예제

- ▶ 다음은 $i=1$, 즉 물건 1에 대해서 배낭의 용량을 1부터 C 까지 늘려 가며 알고리즘을 수행한 결과이다.

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											
3	50	4											

Knapsack 예제

- ▶ $i=2$ 일 때 (즉, 물건 1에 대한 부분문제들의 해는 $i=1$ 일 때 앞에서 이미 계산하였고, 이를 이용하여 물건 2를 고려한다.)
 - ▶ $w=1, 2, 3$ (배낭의 용량이 각각 1, 2, 3kg)일 때, 물건 2를 배낭에 담아보려고 한다. 그러나 $w_2 > w$ 이므로, 즉 물건 2의 무게가 4kg이므로 배낭에 담을 수 없다. 따라서 $K[2, 1]=0, K[2, 2]=0, K[2, 3]=0$ 이다.
 - ▶ $w=4$ (배낭의 용량이 4kg)일 때, 물건 2를 배낭에 담을 수 있다.

$$\begin{aligned}K[2, 4] &= \max\{K[i-1, w], K[i-1, w-w_i]+v_i\} \\&= \max\{K[2-1, 4], K[2-1, 4-4]+40\} \\&= \max\{K[1, 4], K[1, 0]+40\} \\&= \max\{0, 0+40\} \\&= \max\{0, 40\} = 40\end{aligned}$$

Knapsack 예제

- ▶ $w=5$ (배낭의 용량이 5kg)일 때, 물건 2의 무게가 4kg이므로, 역시 배낭에 담을 수 있다. 그러나 이 경우에는 물건 1이 배낭에 담았을 때의 가치와 물건 2를 담았을 때의 가치를 비교하여, 더 큰 가치를 얻는 물건을 배낭에 담는다.

$$\begin{aligned} K[2, 5] &= \max\{K[i-1, w], K[i-1, w-w_i]+v_i\} \\ &= \max\{K[2-1, 5], K[2-1, 5-4]+40\} \\ &= \max\{K[1, 5], K[1, 1]+40\} \\ &= \max\{10, 0+40\} \\ &= \max\{10, 40\} = 40 \text{이다.} \end{aligned}$$

가장큰가치

- ▶ 즉, 물건 1을 배낭에서 빼낸 후 물건 2를 담는 것이므로 그때의 가치가 40이 된다.
- ▶ $w=6, 7, 8$ 일 때, 각각의 경우도 물건 1을 빼내고 물건 2를 배낭에 담는 것이 더 큰 가치를 얻는다. 따라서 각각 $K[2, 6] = K[2, 7] = K[2, 8] = 40$ 이 된다.

Knapsack 예제

- ▶ $w=9$ (배낭의 용량이 9kg)일 때, 물건 2를 배낭에 담아보려고 한다. 그런데 $w_2 < w$ 이므로, 물건 2를 배낭에 담을 수 있다.

$$\begin{aligned} K[2, 9] &= \max\{K[i-1, w], K[i-1, w-w_i]+v_i\} \\ &= \max\{K[2-1, 9], K[2-1, 9-4]+40\} \\ &= \max\{K[1, 9], K[1, 5]+40\} \\ &= \max\{10, 10+40\} \\ &= \max\{10, 50\} = 50 \end{aligned}$$

- ▶ 즉, 이때에는 배낭에 물건 1, 2를 모두 담을 수 있는 것이고, 그때의 가치가 50이 된다는 의미이다.
- ▶ $w=10$ (배낭의 용량이 10kg)일 때, $w_2 < w$ 이므로, $w=9$ 일 때와 마찬가지로 $K[2, 10]=50$ 이고, 물건 1, 2를 배낭에 둘 다 담을 때의 가치인 50을 얻는다는 의미이다.

Knapsack 예제

- ▶ 다음은 $i=2$, 즉 물건 1과 2에 대해서 배낭의 용량을 1부터 C 까지 늘려가며 알고리즘을 수행한 결과이다.

$C=10$

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	0	0	0	0	10	10	10	10	10	10
4	40	$i=2$	0	0	0	0	40	40	40	40	40	50	50
6	30	3	0										
3	50	4											

Knapsack 예제

- ▶ 다음은 $i=3$ 과 $i=4$ 일 때 알고리즘 수행을 마친 결과이다.

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	0	0	0	0	10	10	10	10	10	10
4	40	2	0	0	0	0	40	40	40	40	40	50	50
6	30	3	0	0	0	0	40	40	40	40	40	50	70
3	50	4	0	0	0	50	50	50	50	90	90	90	90

- ▶ 최적해는 $K[4, 10]$ 이고, 그 가치는 물건 2와 4의 가치의 합인 90이다.

시간복잡도

- ▶ 하나의 부분문제에 대한 해를 구할 때의 시간복잡도는 line 5에서 무게를 한 번 비교한 후 line 6에서 1개의 부분문제의 해를 참조하고, line 8에서 2개의 해를 참조한 계산하므로 $O(1)$ 시간이 걸린다.
- ▶ 부분문제의 총 수는 배열 K 의 원소 수인 $n \times C$ 개이다. 여기서 C 는 배낭의 용량이다.
- ▶ Knapsack 알고리즘의 시간복잡도는 $O(1) \times n \times C = O(nC)$ 이다.

배낭 용량: C

nC

응용

- ▶ 배낭 문제는 다양한 분야에서 의사 결정 과정에 활용된다.
 - ▶ 원자재의 버리는 부분을 최소화시키기 위한 자르기/분할
 - ▶ 금융 포트폴리오와 자산 투자의 선택
 - ▶ 암호 생성 시스템(Merkle–Hellman Knapsack Cryptosystem)

5.5 동전 거스름돈

- ▶ 잔돈을 동전으로 거슬러 받아야 할 때, 누구나 적은 수의 동전으로 거스름돈을 받고 싶어한다.
 - ▶ 대부분의 경우 그리디 알고리즘으로 해결할 수 있으나, 해결 못하는 경우도 있다.
 - ▶ 동적 계획 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.
- ▶ 동전 거스름돈 문제
 - ▶ 정해진 동전의 종류가 d_1, d_2, \dots, d_k 이고, 거스름 돈 n 원이다. 단, $d_1 > d_2 > \dots > d_k = 1$ 이라고 하자.
 - ▶ 예를 들어, 우리나라의 동전 종류는 5개로서, $d_1=500, d_2=100, d_3=50, d_4=10, d_5=1$ 이다.

동전 거스름돈

- ▶ 동적 계획 알고리즘을 위한 부분문제
 - ▶ 5.4절의 배낭 문제의 동적 계획 알고리즘을 살펴보면, 배낭의 용량을 1kg 씩 증가시켜 문제를 해결한다.
 - ▶ 여기서 힌트를 얻어서, 동전 거스름돈 문제도 1원씩 증가시켜 문제를 해결한다. 즉, 거스름돈을 배낭의 용량으로 생각하고, 동전을 물건으로 생각하면 이해가 쉬울 것이다.
 - ▶ 부분문제들의 해를 아래와 같이 1차원 배열 C 에 저장하자.
 - ▶ 1원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[1]$
 - ▶ 2원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[2]$
 - ...
 - ▶ j 원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[j]$
 - ...
 - ▶ n 원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[n]$

동전 거스름돈

- ▶ $C[j]$ 를 구하는데 어떤 부분문제가 필요할까? j 원을 거슬러 받을 때 최소의 동전 수를 다음의 동전들 ($d_1=500, d_2=100, d_3=50, d_4=10, d_5=1$)로 생각해 보자.
 - ▶ 500원짜리 동전이 거스름돈 j 원에 필요하면 $(j-500)$ 원의 해, 즉 $C[j-500] = C[j-d_1]$ 에 500원짜리 동전 1개를 추가한다.
 - ▶ 100원짜리 동전이 거스름돈 j 원에 필요하면 $(j-100)$ 원의 해, 즉 $C[j-100] = C[j-d_2]$ 에 100원짜리 동전 1개를 추가한다.
 - ▶ 50원짜리 동전이 거스름돈 j 원에 필요하면 $(j-50)$ 원의 해, 즉 $C[j-50] = C[j-d_3]$ 에 50원짜리 동전 1개를 추가한다.
 - ▶ 10원짜리 동전이 거스름돈 j 원에 필요하면 $(j-10)$ 원의 해, 즉 $C[j-10] = C[j-d_4]$ 에다가 10원짜리 동전 1개를 추가한다.
 - ▶ 1원짜리 동전이 거스름돈 j 원에 필요하면 $(j-1)$ 원의 해, 즉 $C[j-1] = C[j-d_5]$ 에다가 1원짜리 동전 1개를 추가한다.

동전 거스름돈

- ▶ 따라서 $C[j]$ 는 아래와 같이 정의된다.

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

- ▶ 위의 식에서 i 가 1부터 k 까지 변하면서, 즉 $d_1, d_2, d_3, \dots, d_k$ 각각에 대하여 해당 동전을 거스름돈에 포함시킬 경우의 동전 수를 고려하여 최솟값을 $C[j]$ 로 정한다.
- ▶ 단, 거스름돈 j 보다 큰 동전은 고려하지 않는다.

동전 거스름돈 알고리즘

DPCoinChange

입력: 거스름돈 n 원, k 개의 동전의 액면, $d_1 > d_2 > \dots > d_k = 1$

출력: $C[n]$

```
1  for i = 1 to n    C[i]=∞
2  C[0]=0
3  for j = 1 to n {      // j는 1원부터 증가하는 (임시) 거스름돈 액수이고,
                        // j=n이면 입력에 주어진 거스름돈이 된다.
4      for i = 1 to k { // 액면이 가장 높은 동전부터 1원짜리 동전까지
5          if ( $d_i \leq j$ ) and ( $C[j-d_i] + 1 < C[j]$ )
6              C[j]=C[j-d_i]+1
7      }
    }
    return C[n]
```


동전 거스름돈 알고리즘

Line 1	배열 C 의 각 원소를 ∞ 로 초기화 한다. 이는 문제에서 거슬러 받는 최소 동전 수를 구하기 때문이다.
Line 2	$C[0]=0$ 으로 초기화한다. 이는 line 5에서 $C[j-d_i]$ 의 인덱스인 $j-d_i$ 가 0이 되는 경우, 즉 $C[0]$ 이 되는 경우를 위해서이다.
Line 3~6	for-루프에서는 임시 거스름돈 액수 j 를 1원부터 1원씩 증가시키며, line 4~6에서 $\min_{1 \leq i \leq k} \{C[j-d_i] + 1\}$ 을 $C[j]$ 로 정한다. 이를 위해 line 4~6의 for-루프에서는 가장 큰 액면의 동전부터 1원짜리 동전까지 차례로 동전을 고려해보고, 그중에서 최소의 동전 수를 $C[j]$ 로 결정한다. 단, 거스름돈 액수인 j 원보다 크지 않은 동전에 대해서만 고려한다.

DPCoinChange 예제

- ▶ 다음은 $d_1=16, d_2=10, d_3=5, d_4=1$ 이고, 거스름돈 $n=20$ 일 때 DPCoinChange 알고리즘이 수행되는 과정이다.
 - ▶ 각각의 표에서 파란 음영으로 표시된 원소가 $C[j]$ 를 계산하는데 필요한 부분문제들의 해이다.
- ▶ Line 1~2에서 배열 C 를 아래와 같이 초기화시킨다.

j	0	1	2	3	4	5	6	7	8	9	10	...	16	17	18	19	20
C	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	...	∞	∞	∞	∞	∞

DPCoinChange 예제

- ▶ 임의 거스름돈 j 원 ($=1, 2, 3, 4$)은 1원짜리 동전($d_4=1$)밖에 고려할 동전이 없으므로, 각 j 에서 1을 뺀, 즉 $(j-1)$ 의 해인 $(C[j-1]+1)$ 이 $C[j]$ 가 된다.
- ▶ 따라서 $i=4$ (1원짜리 동전)일 때의 line 5의 if-조건인 $(1 \leq j)$ 가 '참'이고, $(C[j-1]+1 < \infty)$ 도 '참'이 되어 각각 아래와 같이 $C[j]$ 가 결정된다.
- ▶ $j=1: C[1] = C[j-1]+1 = C[1-1]+1 = C[0]+1 = 0+1 = 1$

j	0	1
	0	∞

 \Rightarrow

j	0	1
	0	1

DPCoinChange 예제

- ▶ $j=2: C[2] = C[j-1] + 1 = C[2-1] + 1 = C[1] + 1 = 1 + 1 = 2$

j	1	2
	1	∞

 \Rightarrow

j	1	2
	1	2

- ▶ $j=3: C[3] = C[j-1] + 1 = C[3-1] + 1 = C[2] + 1 = 2 + 1 = 3$

j	2	3
	2	∞

 \Rightarrow

j	2	3
	2	3

- ▶ $j=4: C[4] = C[j-1] + 1 = C[4-1] + 1 = C[3] + 1 = 3 + 1 = 4$

j	3	4
	3	∞

 \Rightarrow

j	3	4
	3	4

DPCoinChange 예제

- ▶ $j=5$ 일 때 (즉 임시 거스름돈이 5원),
 - ▶ $i=3$ (5원짜리 동전)에 대해서, line 5의 if-조건인 $(d_3 \leq 5)$ 가 '참'이고, $(C[j-d_i]+1 < C[j]) = (C[5-5]+1 < C[5]) = (C[0]+1 < \infty) = (0+1 < \infty)$ 이므로 '참'이 되어 ' $C[j] = C[j-d_i]+1$ '가 수행된다. 따라서 $C[5] = C[5-5]+1 = C[0]+1 = 0+1 = 1$ 이 된다. 즉, $C[5]=1$ 이다.
 - ▶ $i=4$ (1원짜리 동전)일 때는 line 5의 if-조건인 $(d_4 \leq 5)$ 는 '참'이나 $(C[j-d_i]+1 < C[j]) = (C[5-1]+1 < C[5]) = (C[4]+1 < C[5]) = (4+1 < 1) = (5 < 1)$ 이 '거짓'이 되어 $C[5]$ 는 변하지 않고 그대로 1을 유지한다. 즉, 1원짜리 동전으로 거스름돈을 주려 하면 오히려 동전 수가 늘어나기 때문이다.

j	0	1	2	3	4	5
	0	1	2	3	4	∞

⇒

j	0	1	2	3	4	5
	0	1	2	3	4	1

+1
↓ 선택 +4
↖
거짓

DPCoinChange 예제

- ▶ $j=6, 7, 8, 9$ 이고, $i=3$ (5원짜리 동전)일 때, 각각 다음과 같이 수행된다.
 - ▶ $C[6]=C[j-5]+1=C[6-5]+1=C[1]+1=1+1=2$
 - ▶ $C[7]=C[j-5]+1=C[7-5]+1=C[2]+1=2+1=3$
 - ▶ $C[8]=C[j-5]+1=C[8-5]+1=C[3]+1=3+1=4$
 - ▶ $C[9]=C[j-5]+1=C[9-5]+1=C[4]+1=4+1=5$
- ▶ $i=4$ (1원짜리 동전)일 때 line 5의 if-조건 $(C[j-d_i]+1 < C[j]) = (C[j-1]+1 < C[j])$ 이 각각의 j 에 대해서 $(1+1) < 2, (2+1) < 3, (3+1) < 4, (4+1) < 5$ 로서 '거짓'이 되어 $C[j]$ 는 변경되지 않는다.

j	0	1	2	3	4	5	6	7	8	9
C	0	1	2	3	4	1	∞	∞	∞	∞
	0	1	2	3	4	1	2	∞	∞	∞
	0	1	2	3	4	1	2	3	∞	∞
	0	1	2	3	4	1	2	3	4	∞
	0	1	2	3	4	1	2	3	4	5

DPCoinChange 예제

- ▶ $j=10$ 일 때 (즉 임시 거스름돈이 10원),
 - ▶ $i=2$ (10원짜리 동전)일 때, line 5의 if-조건인 $(d_i \leq j) = (10 \leq 10)$ 은 ‘참’이고, $(C[j-d_i] + 1 < C[j]) = (C[0] + 1 < C[10]) = (0 + 1 < \infty)$ 이 ‘참’이 되어 ‘ $C[j] = C[j-d_i] + 1$ ’이 수행된다. 따라서 $C[10] = C[10-10] + 1 = C[0] + 1 = 0 + 1 = 1$ 이다. 즉, $C[10]=1$ 이다.
 - ▶ $i=3$ (5원짜리 동전)일 때, line 5의 if-조건인 $(d_i \leq j) = (5 < 10)$ 는 ‘참’이나, $(C[j-d_i] + 1 < C[j]) = (C[10-5] + 1 < C[10]) = (C[5] + 1 < C[10]) = (1 + 1 < 1)$ 이 ‘거짓’이므로 $C[10]$ 은 변하지 않는다. 즉, 5원짜리 2개보다는 10원짜리 1개가 낫다.
 - ▶ $i=4$ (1원짜리 동전)일 때, line 5의 if-조건인 $(d_i \leq j) = (1 < 10)$ 는 ‘참’이나, $(C[j-d_i] + 1 < C[j]) = (C[10-1] + 1 < C[10]) = (C[9] + 1 < C[10]) = (5 + 1 < 1)$ 이 ‘거짓’이므로 $C[10]$ 은 변하지 않고 그대로 1을 유지한다.

j	0	1	2	3	4	5	6	7	8	9	10
C	0	1	2	3	4	1	2	3	4	5	1

DPCoinChange 예제

- ▶ $j=11$ 일 때,
 - ▶ $i=2$ (10원짜리 동전)일 때, $C[11] = C[j-10]+1 = C[1]+1 = 1+1 = 2$.
 - ▶ $i=3$ (5원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-5]+1 < C[j]) = (C[11-5]+1 < C[11]) = (C[6]+1 < C[11]) = (2+1 < 2) = (3 < 2)$ 이 '거짓'이므로 $C[11]$ 은 변하지 않는다.
 - ▶ $i=4$ (1원짜리 동전)일 때, line 5의 if-조건인 $(C[11-1]+1 < 2) = (2 < 2)$ 이 '거짓'이므로 $C[11]$ 은 변하지 않는다.

j	0	1	2	3	4	5	6	7	8	9	10	11
C	0	1	2	3	4	1	2	3	4	5	1	2

DPCoinChange 예제

- ▶ $j=12, 13, 14$ 일 때,
 - ▶ $C[11]$ 에 비해서 1원짜리 동전만 하나씩 추가된다.
 - ▶ $C[12] = 3$
 - ▶ $C[13] = 4$
 - ▶ $C[14] = 5$

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5

DPCoinChange 예제

- ▶ $j=15$ 일 때,
 - ▶ $i=2$ (10원짜리 동전)일 때, $C[15] = C[j-10] + 1 = C[5] + 1 = 1 + 1 = 2$
 - ▶ $i=3$ (5원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i] + 1 < C[j]) = (C[j-5] + 1 < C[j]) = (C[15-5] + 1 < C[15]) = (C[10] + 1 < C[15]) = (1 + 1 < 2) = (2 < 2)$ 이 '거짓'이므로 $C[15]$ 는 변하지 않는다.
 - ▶ $i=4$ (1원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i] + 1 < C[j]) = (C[j-1] + 1 < C[j]) = (C[15-1] + 1 < C[15]) = (C[14] + 1 < C[15]) = (5 + 1 < 2) = (6 < 2)$ 이 '거짓'이므로 $C[15]$ 는 변하지 않는다.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2

DPCoinChange 예제

- ▶ $j=16$ 일 때,
 - ▶ $i=1$ (16원짜리 동전)일 때, $C[16] = C[j-16]+1 = C[0]+1 = 0+1 = 1$.
 - ▶ $i=2$ (10원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-10]+1 < C[j]) = (C[16-10]+1 < C[16]) = (C[6]+1 < C[16]) = (3 < 1)$ 이 '거짓'이므로 $C[16]$ 는 변하지 않는다.
 - ▶ $i=3$ (5원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-5]+1 < C[j]) = (C[16-5]+1 < C[16]) = (C[11]+1 < C[16]) = (3 < 1)$ 이 '거짓'이므로 $C[16]$ 은 변하지 않는다.
 - ▶ $i=4$ (1원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-1]+1 < C[j]) = (C[16-1]+1 < C[16]) = (C[15]+1 < C[16]) = (3 < 1)$ 이 '거짓'이므로 $C[16]$ 은 변하지 않는다.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	1

DPCoinChange 예제

- ▶ $j=17, 18, 19$ 일 때,
 - ▶ $C[16]$ 에 비해서 1원짜리 동전만 하나씩 추가된다.
 - ▶ $C[17] = 2$
 - ▶ $C[18] = 3$
 - ▶ $C[19] = 4$

DPCoinChange 예제

- ▶ $j=20$ 일 때,
 - ▶ $i=1$ (16원짜리 동전)일 때, $C[20] = C[j-16]+1 = C[4]+1 = 4+1 = 5$.
 - ▶ $i=2$ (10원짜리 동전)일 때, line 5의 if-조건에서 $C[j-d_i]+1 = C[j-10]+1 = C[10]+1 = 1+1 = 2$ 이므로 현재 $C[20]$ 의 값인 5보다 작다. 따라서 if-조건이 '참'이 되어 $C[20]=2$ 가 된다.
 - ▶ $i=3$ (5원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-5]+1 < C[j]) = (C[20-5]+1 < C[20]) = (C[15]+1 < C[20]) = (3 < 2)$ 이 '거짓'이므로 $C[20]$ 은 변하지 않는다.
 - ▶ $i=4$ (1원짜리 동전)일 때, line 5의 if-조건인 $(C[j-d_i]+1 < C[j]) = (C[j-1]+1 < C[j]) = (C[20-1]+1 < C[20]) = (C[19]+1 < C[20]) = (5 < 2)$ 이 '거짓'이므로 $C[20]$ 은 변하지 않는다.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	1	2	3	4	2

DPCoinChange 예제

- ▶ 따라서 거스름돈 20원에 대한 최종해는 $C[20]=2$ 개의 동전이다.
- ▶ (비교) 4.1절의 그리디 알고리즘은 20원에 대해 16원짜리 동전을 먼저 ‘욕심내어’ 취하고, 4원이 남게 되어, 1원짜리 4개를 취하여, 모두 5개의 동전이 해라고 답한다.

시간복잡도

- ▶ DPChange 알고리즘의 시간복잡도는 $O(nk)$ 인데 이는 거스름돈 j 가 1원부터 n 원까지 변하며, 각각의 j 에 대해서 최악의 경우 k 개의 모든 동전 (d_1, d_2, \dots, d_k) 를 1번씩 고려하기 때문이다.