6. 동적 계획 알고리즘 2

한국외국어대학교
고 석 훈

목차

6.3 편집 거리 문제

- 문서 편집기를 사용하여 문서를 작성하는 중에 스트링 S를 수정하여 다른 스트링 T로 변환시키고자 할 때, 삽입(insert), 삭제(delete), 대체(substitute) 연산이 사용된다.

- 이때, S를 T로 변환시키는데 필요한 최소의 편집 연산 횟수를 **편집 거리(Edit Distance)**라고 한다.

- 예를 들어, 'strong'을 'stone'으로 편집하는 경우,

- 편집 방법 1:
  - 's'와 't'를 그대로 사용, 'o'를 삽입, 'r'과 'o'를 삭제, 그 다음 'n'을 그대로 사용, 마지막 'g'를 'e'로 대체
  - 총 4회의 편집 연산 수행

s	**t**		**r**	**o**	**n**	**g**
⇩	⇩	삽입	삭제	삭제	⇩	대체
s	**t**	**o**			**n**	**e**

- 편집 방법 2:
  - 's'와 't'는 그대로 사용, 'r'을 삭제하고,
    'o'와 'n'을 그대로 사용, 'g'를 'e'로 대체
  - 총 2회의 편집 연산 수행, 이는 최소 편집 횟수이다.

s	**t**	**r**	**o**	**n**	**g**
⇩	⇩	삭제	⇩	⇩	대체
s	**t**		**o**	**n**	**e**

- 이처럼 어떤 연산을 어느 문자에 수행하는가에 따라서
  편집 연산 횟수가 달라진다.

편집 거리 문제의 부분 문제

- 편집 거리 문제를 동적 계획 알고리즘으로 해결하려면 부분 문제들을 구분해야 한다.
  - 'strong'을 'stone'으로 편집하려는데, 만일 각 접두부 (prefix)에 대한 편집거리를 알고 있으면, 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.

		1	2	3	4		
S	**=**	**s**	**t**	**r**	**o**	**n**	**g**
T	**=**	**s**	**t**	**o**	**n**	**e**	
		1	2	3			

- 부분 문제를 정의하기 위해서
스트링 S와 T의 길이를 각각 m과 n이라 하고,
S와 T의 각 문자를 다음과 같이 s_i와 t_j라고 하자.
단, $i = 1, 2, \cdots, m$이고, $j = 1, 2, \cdots, n$이다.

$$\begin{aligned} S &= s_1 s_2 s_3 \cdots s_m \\ T &= t_1 t_2 t_3 \cdots t_n \end{aligned}$$

- 부분 문제 $E[i, j]$는 S의 접두부의 i개 문자를 T의 접두부 j개 문자로 변환시키는데 필요한 최소 편집 연산 횟수, 즉, 편집 거리를 의미한다.
  - 예를 들어, 'strong'을 'stone'으로 편집하는 경우,
'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 문제의 예

- 다음 예제에 대해 처음 몇 개의 부분 문제의 편집 거리를 계산해 보자.

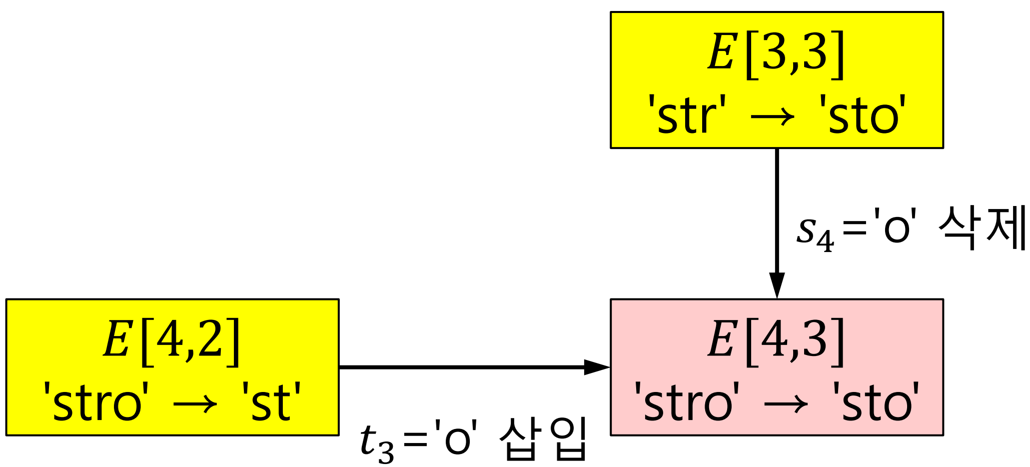
		1	2	3	4	5	6
S	**=**	**s**	**t**	**r**	**o**	**n**	**g**
T	**=**	**s**	**t**	**o**	**n**	**e**	
		1	2	3	4	5	

첫 번째 부분의 예

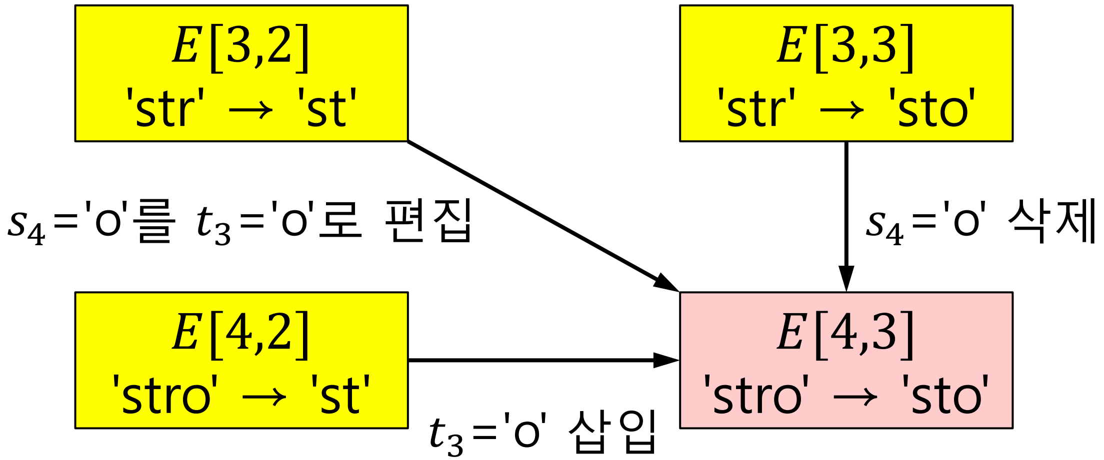
- $s_1 \rightarrow t_1$ ['s' → 's'] 부분 문제 $E[1,1] = ?$
  - $E[1,1] = 0$이다. 왜냐하면 $s_1 = t_1 =$'s' 이기 때문이다.
- $s_1 \rightarrow t_1 t_2$ ['s' → 'st'] 부분 문제 $E[1,2] = ?$
  - $E[1,2] = 1$이다. 왜냐하면 $s_1 = t_1 =$'s' 이고,
    't'를 삽입하는데 1회의 연산이 필요하기 때문이다.
- $s_1 s_2 \rightarrow t_1$ ['st' → 's'] 부분 문제 $E[2,1] = ?$
  - $E[2,1] = 1$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' → 'st'] 부분 문제 $E[2,2] = ?$
  - $E[2,2] = 0$이다. 왜냐하면 $s_1 = t_1 =$'s' 이고, $s_2 = t_2 =$ 't' 이기 때문이다.
    이 경우에는 $E[1,1]$=0이라는 결과를 미리 계산하여 놓았고,
    $s_2 = t_2 =$ 't' 이므로, $E[2,2] = [1,1] + 0 = 0$인 것이다.

일반적인 경우의 예

- $s_1 s_2 s_3 s_4 \rightarrow t_1 t_2 t_3$ ['stro' → 'sto'] 부분 문제 $E[4,3] = ?$
  - $s_1 s_2 s_3 s_4 \rightarrow t_1 t_2$ ['stro' → 'st'] 부분 문제 $E[4,2]$의 해를 알면, t_3='o'를 삽입하면 되므로, 편집 연산 횟수는 $E[4,2] + 1$이 된다.
  - $s_1 s_2 s_3 \rightarrow t_1 t_2 t_3$ ['str' → 'sto'] 부분 문제 $E[3,3]$의 해를 알면, s_4='o'를 삭제하면 되므로, 편집 연산 횟수는 $E[3,3] + 1$이 된다.



- $s_1 s_2 s_3 \rightarrow t_1 t_2$ ['str' → 'st'] 부분 문제 $E[3,2]$의 해를 알면, s_4='o'를 t_3='o'로 편집하는데 필요한 연산을 계산하면 된다. 이때, $s_4 = t_3$ 이므로 편집 연산 횟수는 $E[3,2] + 0$ 이 된다. (만일, $s_4 \neq t_3$ 이었다면 s_4를 t_3로 대체를 해야 하므로 편집 연산 횟수는 $E[3,2] + 1$ 이 될 것이다.)



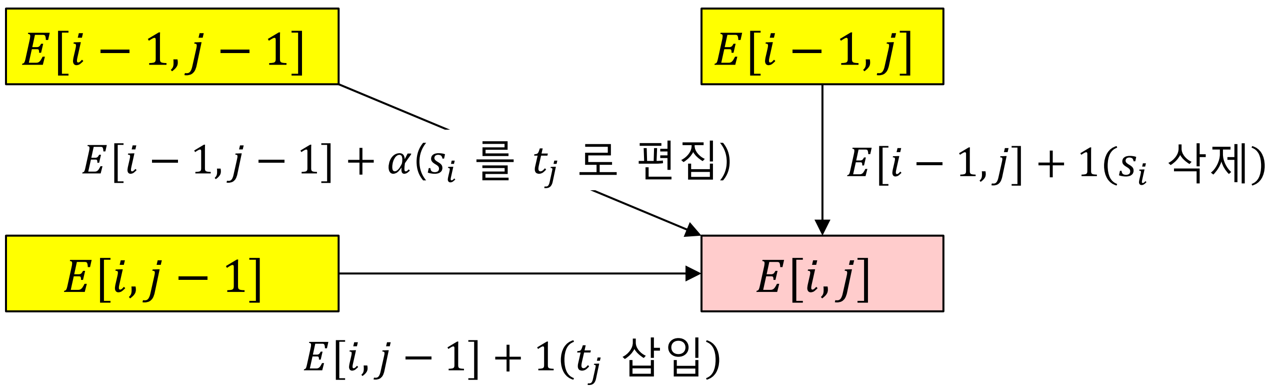
- 따라서 $E[4,3]$의 편집 거리는 위의 3가지 부분 문제들의 해, 즉, $E[4,2], E[3,3], E[3,2]$의 편집 거리로부터 계산할 수 있다.

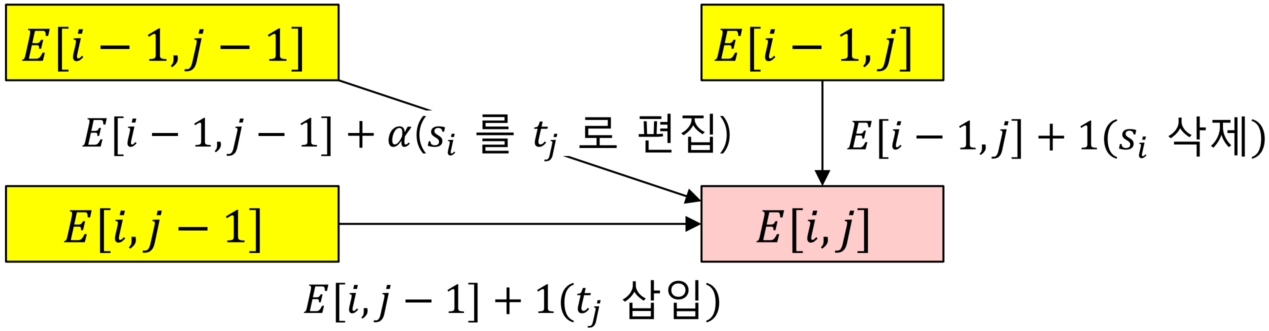
  즉, $E[4,2] = 2, E[3,3] = 1, E[3,2] = 1$이므로,
  $(2+1), (1+1), (1+0)$ 중에서 최소값인 $(1+0)$이
  $E[4,3]$의 편집 거리가 된다.

$E(i,j)$	T	ε	s	t	o
S	$i \backslash j$	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 문제의 함축적 순서

- 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 부분 문제간의 함축적 순서는 다음과 같다.





- $E[i-1, j]$는 $s_1 s_2 \dots s_{i-1} \rightarrow t_1 t_2 \dots t_j$의 해이므로 s_i 삭제 연산을 추가하면 $s_1 s_2 \dots s_i \rightarrow t_1 t_2 \dots t_j$의 해가 되므로 $E[i, j] = E[i-1, j] + 1$이 될 수 있다.

- $E[i, j-1]$는 $s_1 s_2 \dots s_i \rightarrow t_1 t_2 \dots t_{j-1}$의 해이므로 t_j 삽입 연산을 추가하면 $s_1 s_2 \dots s_i \rightarrow t_1 t_2 \dots t_j$의 해가 되므로 $E[i, j] = E[i, j-1] + 1$이 될 수 있다.

- $E[i-1, j-1]$는 $s_1 s_2 \dots s_{i-1} \rightarrow t_1 t_2 \dots t_{j-1}$의 해이므로 s_i와 t_j의 연산을 추가하면 $s_1 s_2 \dots s_i \rightarrow t_1 t_2 \dots t_j$의 해가 되므로 $E[i, j] = E[i-1, j-1] + \alpha$가 될 수 있다.
  이때 s_i와 t_j가 같으면 연산이 필요 없으므로 $\alpha = 0$ 이 되고,
  s_i와 t_j가 다르면 대체 연산이 필요하기 때문에 $\alpha = 1$ 이 된다.

규칙 도출

- 따라서 위의 3가지 경우 중에서
  가장 작은 값을 $E[i,j]$의 해로서 선택한다. 즉,

$$E[i,j] = \min\{\, E[i,j-1]+1, E[i-1,j]+1, E[i-1,j-1]+\alpha \,\}$$
$$\text{단}, \alpha = 1\ if\ s_i \neq t_j, else\ \alpha = 0$$

- 위의 식을 위해서 $E[0,0], E[1,0], E[2,0], \cdots, E[m,0]$과 $E[0,1], E[0,2], \cdots, E[0,n]$을 아래와 같이 초기화한다.

$E[i,j]$	T	ε	t_1	t_2	t_3	$\cdots$	t_n
S	$i \backslash j$	0	1	2	3	$\cdots$	n
ε	0	0	1	2	3	$\cdots$	n
s_1	1	1					
s_2	2	2					
s_3	3	3					
$\vdots$	$\vdots$	$\vdots$					
s_m	m	m					

- 배열 E의 0번 행의 $0, 1, 2, \cdots, n$ 초기값은 S가 ε(공백 스트링)인 상태에서 T의 문자를 좌에서 우로 하나씩 삽입해가는 연산 횟수를 나타낸다.
  - $E[0,0] = 0$, T의 첫 문자를 만들기 이전이므로, 아무런 연산이 필요 없다.
  - $E[0,1] = 1$, T의 첫 문자를 만들기 위해 't_1'을 삽입
  - $E[0,2] = 2$, T의 처음 2문자를 만들기 위해 '$t_1 t_2$'를 각각 삽입
  - $\cdots$
  - $E[0,n] = 5$, T를 만들기 위해 '$t_1 t_2 t_3 \cdots t_n$'을 각각 삽입
- 배열 E의 0번 열의 $0, 1, 2, \cdots, m$ 초기값은 T를 ε(공백 스트링)으로 만들기 위해서, S의 문자를 위에서 아래로 하나씩 삭제하는 연산 횟수를 나타낸다.
  - $E[0,0] = 0$, S의 첫 문자를 지우기 이전이므로, 아무런 연산이 필요 없다.
  - $E[1,0] = 1$, S의 첫 문자 's_1'을 삭제해야 T가 ε 가 된다.
  - $E[2,0] = 2$, S의 처음 2 문자 '$s_1 s_2$'를 삭제해야 T가 ε 가 된다.
  - $\cdots$
  - $E[m,0] = m$, T의 모든 문자를 삭제해야 T가 ε 가 된다.

편집 거리 알고리즘

$\boldsymbol{EditDistance}$

입력: 스트링 S, T, 단, S와 T의 길이는 각각 m과 n이다.

출력: S를 T로 변환하는 편집 거리, $E[m,n]$

$for\ i = 0\ to\ m\ E[i,0] = i$ // 0번 열의 초기화

$for\ j = 0\ to\ n\ E[0,j] = j$ // 0번 행의 초기화

$for\ i = 1\ to\ m$

$\quad for\ j = 1\ to\ n$

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

$return\ E[m,n]$

EditDistance 알고리즘 실행 예

- EditDistance 알고리즘으로 'strong'을 'stone'으로 바꾸는데 필요한 편집 거리를 계산한 결과인 배열 E이다.
  - 붉은색 음영으로 표시된 원소가 계산되는 과정을 상세히 살펴보자.

$E(i,j)$	T	ε	s	t	o	n	e
S	$i \backslash j$	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

- $E[1,1] \; = \min\{E[1,0] + 1, E[0,1] + 1, E[0,0] + \alpha\}$
  $= \min\{(1 + 1), (1 + 1), (0 + 0)\} = 0$
  - $E[1,0] + 1 = 2$: S의 첫 문자를 삭제하여 $E[1,0] = 1$인 상태에서 T의 첫 문자 t_1='s'를 삽입한다는 의미
  - $E[0,1] + 1 = 2$: T의 첫 문자인 t_1을 삽입하여 $E[0,1] = 1$인 상태에서
    S의 첫 문자인 s_1='s'를 삭제한다는 의미
  - $E[0,0] + \alpha = 0 + 0 = 0$:$s_1$과 t_1이 같기 때문에 $\alpha = 0$
  - 위의 3가지 경우의 값 중에서 최소값인 0이 $E[1,1]$이 된다.

$E(i,j)$	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				
t	2						

- $E[2,2] = \min\{E[2,1]+1, E[1,2]+1, E[1,1]+\alpha\}$
  $= \min\{(1+1),(1+1),(0+0)\} = 0$
  - 현재 T의 첫 문자 's'가 만들어져 있는 상태에서 s_2와 t_2가 't'로 같기 때문에 아무런 연산 없이 'st'가 만들어진다.

$E(i,j)$	T	ε	s	t	o	n	e
S	$i \backslash j$	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			
r	3						
o	4						
n	5						
g	6						

- $E[3,2] = \min\{E[3,1]+1, E[2,3]+1, E[2,2]+\alpha\}$
  $= \min\{(2+1), (0+1), (1+1)\} = 1$
  - 현재 T의 처음 2문자 'st'가 만들어져 있는 상태에서 S의 3번째 문자인 'r'을 삭제한다는 의미이다.

$E(i,j)$	T	ε	s	t	o	n	e
S	$i \backslash j$	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			
o	4						
n	5						
g	6						

- $E[4,3] \;=\; \min\{E[4,2]+1, E[3,3]+1, E[3,2]+\alpha\}$
  $\;=\; \min\{(2+1),(1+1),(1+0)\} \;=\; 1$
  - 현재 T의 처음 2문자 'st'가 만들어져 있는 상태에서 s_4와 t_3가 'o'로 같기 때문에 아무런 연산 없이 'sto'가 만들어진다.

$E(i,j)$	T	ε	s	t	o	n	e
S	$i \backslash j$	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		
n	5						
g	6						

- $E[5,4] = \min\{E[5,3]+1, E[4,4]+1, E[4,3]+\alpha\}$
  $= \min\{(2+1),(1+1),(1+0)\} = 1$
  - 현재 T의 처음 3문자 'sto'가 만들어져 있는 상태에서 s_5와 t_4가 'n'으로 같기 때문에 아무런 연산 없이 'ston'이 만들어진다.

$E(i,j)$	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	
g	6						

- $E[6,5] \;=\; \min\{E[6,4]+1, E[5,5]+1, E[5,4]+\alpha\}$
  $\;=\; \min\{(2+1),(2+1),(1+1)\} \;=\; 2$
  - 현재 T의 처음 4문자 'ston'가 만들어져 있는 상태에서 s_6='g'가 t_5='e'로 대체되어 'stone'이 만들어진다.

$E(i,j)$	T	ε	s	t	o	n	e
S	$i \backslash j$	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

시간복잡도

- EditDistance 알고리즘의 시간복잡도
  - 총 부분 문제의 수는 배열 E의 원소 수인 $m \times n$ 이다.
    여기서, m과 n은 두 스트링 각각의 길이이다.
  - 각 부분 문제를 계산하기 위해
    주위의 3개의 부분 문제들의 해의 최소값을 구한다.
  - 따라서 EditDistance 알고리즘의 시간복잡도는
    이다.

편집거리 문제의 응용분야

- 두개의 스트링 사이의 편집 거리가 작으면, 두 스트링이 서로 유사하다고 볼 수 있으므로, 생물 정보 공학 및 의학 분야에서 두 개의 유전자가 얼마나 유사한가를 측정하는데 활용된다.
- 그 외에도 철자 오류 검색, 광학 문자 인식 보정 시스템, 자연어 번역 등에 활용된다.

6.4 배낭(Knapsack) 문제

- 배낭(Knapsack) 문제
  - 한정된 무게의 물건을 담을 수 있는 배낭과
    고유한 무게와 가치를 가지는 n개의 물건이 있을 때,
    최대의 가치를 갖도록 배낭에 넣을 물건을 정하는 문제

- 두 가지 배낭 문제
  -
    물건을 쪼갤 수 있어서, 배낭에 물건을 부분적으로 넣는 것을 허용
  -
    물건을 나눌 수 없어서, 배낭에 하나의 물건을 통째로 넣어야 함

0/1 배낭 문제

- 배낭 (Knapsack) 문제는 용량이 C인 배낭과 n개의 물건과 각 물건 i의 무게 w_i와 가치 v_i가 주어졌을 때, 배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제이다.
  단, 배낭에 담은 물건의 무게의 합이 C를 초과하지 말아야 하고, 각 물건은 1개씩만 있다.

- 배낭 문제를 해결하는 아이디어
  - 어떤 물건에 대해 그 물건을 배낭에 담지 않는 경우와 배낭에 있는 다른 물건을 꺼내고 그 물건을 담는 경우를 비교하여 배낭에 들어가는 물건의 가치가 큰 쪽을 선택한다.
  - 동적 계획 알고리즘으로 문제를 해결하기 위해 차례 차례 고려하는 물건의 개수를 늘려가며, (임시) 배낭의 크기를 1부터 실제 배낭의 용량까지 늘려가며, 각 물건에 대한 부분해를 구하는 방식으로 최종해를 구한다.

배낭 문제의 부분 문제

- 배낭 문제의 부분 문제를 아래와 같이 정의할 수 있다.

  $K[i, w] = 1 \sim i$ 까지의 물건에 대해,
  (임시) 배낭의 용량이 w일 때의 최대 가치
  단, $i = 1, 2, 3, \cdots, n$ 이고, $w = 1, 2, 3, \cdots, C$ 이다.

  - 그러므로 문제의 최적해는 $K[n, C]$이다.
  - 배낭의 용량이 C이지만, 임시 배낭의 용량을 1부터 C까지 1씩 증가시키며 부분 문제를 풀어나간다. 따라서, C의 값이 매우 크면, 알고리즘 수행시간이 매우 길어지는 단점이 있다.

Knapsack 알고리즘

Knapsack

입력: 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 w_i와 가치 v_i,

단, $i = 1, 2, \cdots, n$

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. $\quad for\ w = 1\ to\ C\ \{$ // w는 배낭의 (임시) 용량
5. $\quad\quad if\ (w_i > w)$ // 물건 i의 무게가 배낭 용량을 초과하는 경우
6. $\quad\quad\quad K[i, w] = K[i-1, w]$
7. $\quad\quad else$ // 물건 i의 무게가 배낭에 담을 수 있는 경우
8. $\quad\quad\quad K[i, w] = \max\{\ K[i-1, w],\ K[i-1, w-w_i] + v_i\ \}$

$\quad\}$

$\}$

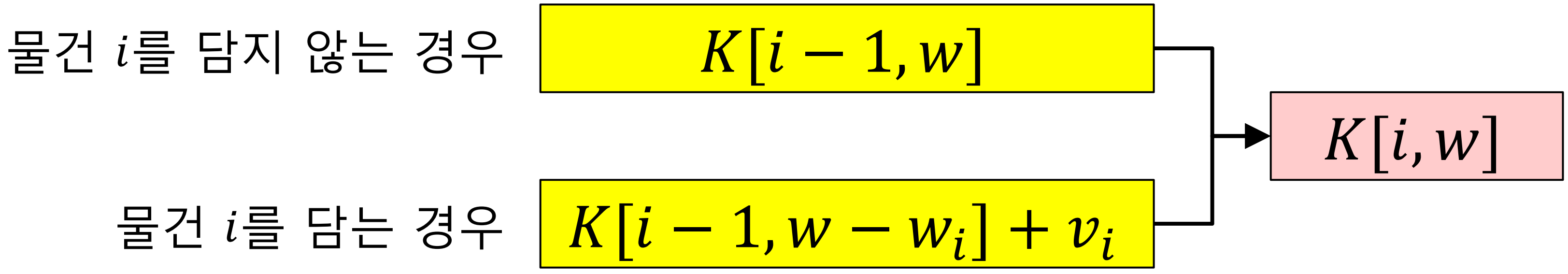
9. $return\ K[n, C]$

- Line 1: 2차원 배열 K의 0번 열을 0으로 초기화
  - 배낭의 용량이 0일 때, 어떤 물건도 배낭에 담을 수 없으므로 최대 가치는 0으로 초기화 한다.
- Line 2: 0번 행의 각 원소를 0으로 초기화
  - 물건 0이란 어떤 물건도 배낭에 담지 않은 상태이므로 배낭의 용량과 관계없이 최대 가치는 0으로 초기화 한다.

- Line 3~8: 배낭에 물건 채우기
  - 물건을 1에서 n까지 하나씩 고려하여
    배낭의 용량을 1에서 C까지 증가시키며 다음을 수행한다.
- Line 5~6:
  - 현재 배낭에 담아보려고 고려하는 물건 i의 무게 w_i가
    배낭의 용량 w보다 크면 물건 i는 무조건 배낭에 담을 수 없다.
  - 그러므로, 물건 i까지 고려했을 때의 최대 가치 $K[i,w]$는
    물건 $(i-1)$까지 고려했을 때의 최대 가치 $K[i-1,w]$가 된다.

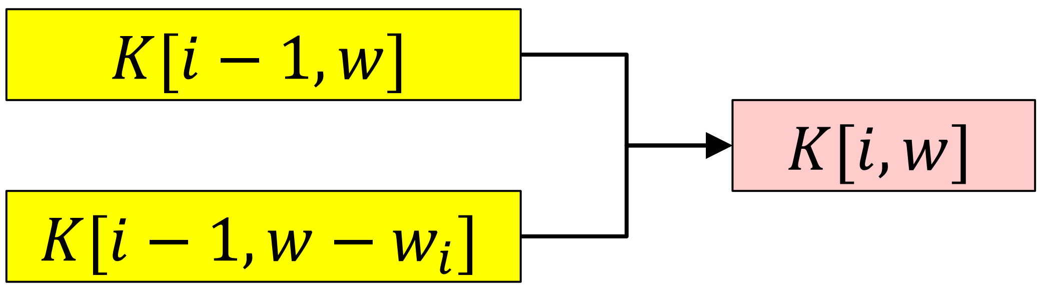
- Line 7~8:
  - 현재 고려하는 물건 i의 무게 w_i가 현재 배낭의 용량 w보다 작거나 같으면 물건 i를 배낭에 담을 수 있는 상태가 된다.
  - 그러나, 물건 i를 배낭에 담으면 배낭 무게가 w를 초과하기 때문에 배낭에 들어있는 다른 물건을 꺼내야 할 수도 있다.
  - 이런 상황을 반영하여 물건 i를 배낭에 담지 않는 경우와 담는 경우의 최대 가치를 비교하여 더 큰 값을 $K[i,w]$로 정한다.

- Line 8: 물건 i를 담을 수 있는 경우, 최대 가치 $K[i,w]$ 계산
  - 물건 i를 배낭에 담지 않는 경우, $K[i,w] = K[i-1,w]$가 된다.
  - 물건 i를 배낭에 담는 경우, 배낭 용량 w에서 물건 i의 무게 w_i를 뺀 용량$(w-w_i)$에서 $(i-1)$까지의 물건을 고려했을 때의 최대 가치인 $K[i-1,w-w_i]$와 물건 i의 가치 v_i의 합이 $K[i,w]$가 된다.
  - 위의 두 가지 경우 중에 큰 값을 $K[i,w]$로 정한다.



부분 문제간의 함축적 순서

- 배낭 문제의 부분 문제간의 함축적 순서는 다음과 같다. 즉, 2개의 부분 문제 $K[i-1, w-w_i]$와 $K[i-1, w]$가 미리 계산되어 있어야만 $K[i, w]$를 계산할 수 있다.



Knapsack 알고리즘 수행과정

- 배낭의 용량 C=10kg이고, 각 물건의 무게와 가치는 다음과 같다.

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

- Line 1~2: 배열의 0번 행과 0번 열의 값을 0으로 초기화한다.

배낭용량 w (C=10kg)			0	1	2	3	4	5	6	7	8	9	10
물건 i	가치 v_i	무게 w_i	0	0	0	0	0	0	0	0	0	0	0
1	10	5	0										
2	40	4	0										
3	30	6	0										
4	50	3	0										

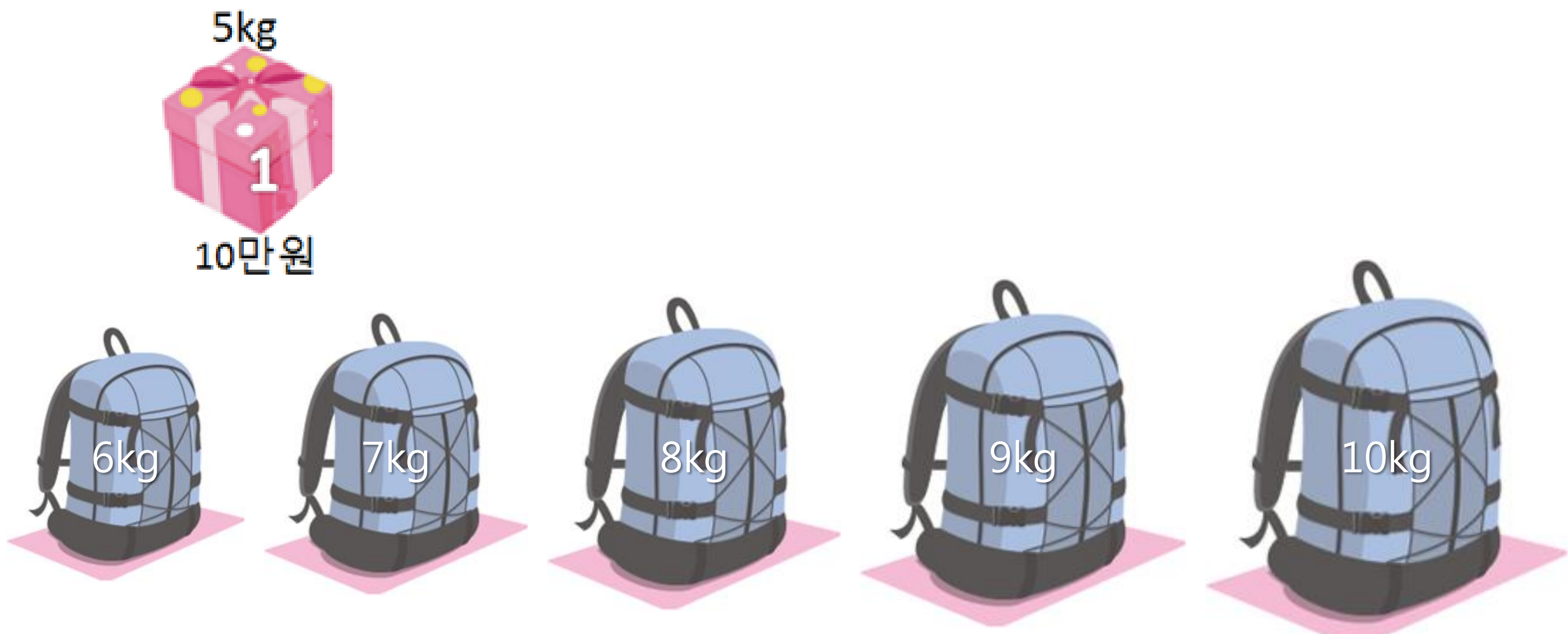
- Line 3~4: 물건 번호 $i = 1{\sim}4$까지 하나씩 반복하고,
배낭의 용량 $w = 1{\sim}10$으로 증가하며 배낭에 물건을 담는다.

- $i = 1$일 때 (즉, 물건 1만을 고려한다.)
  - 배낭 용량 $w = 1kg$ 일 때, 물건 1($w_1 = 5kg$)을 배낭에 담으려 한다. 그러나, $w_1 > w$ 이므로, 물건 1을 배낭에 담을 수 없다. 따라서, $K[1,1] = K[i-1, w] = K[1-1,1] = K[0,1] = 0$ 이다.
  - 배낭 용량 $w = 2, 3, 4kg$일 때도 역시 $w_1 > w$ 이므로, 물건 1을 배낭에 담을 수 없다. 따라서, $K[2,1] = K[1,3] = K[1,4] = 0$ 이다.

- 배낭 용량 $w = 5kg$ 일 때, 물건 1($w_1 = 5kg$)을 배낭에 담으려 한다. 이번에는 $w_1 = w$ 이므로 물건 1을 배낭에 담을 수 있다. 따라서

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

■ 배낭 용량 $w = 6, 7, 8, 9, 10kg$ 일 때도
각각의 경우 $w_1 < w$ 이므로 물건 1을 배낭에 담을 수 있다.
따라서 $K[1,6] = K[1,7] = K[1,8] = K[1,9] = K[1,10] = 10$ 이다.



- 다음은 물건 1에 대해 배낭의 용량을 1에서 C까지 늘려가며 알고리즘을 수행한 결과이다.

배낭용량 w (C=10kg)			0	1	2	3	4	5	6	7	8	9	10
물건 i	가치 v_i	무게 w_i	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										
3	30	6	0										
4	50	3	0										

- $i = 2$일 때 ($i = 1$일 때 구한 물건 1에 대한 부분 문제들의 해를 이용하여 물건 1과 물건 2에 대한 부분해를 구한다.)
  - 배낭 용량 $w = 1, 2, 3kg$ 일 때, 물건 2($w_2 = 4kg$)을 배낭에 담으려 한다. 그러나 $w_2 > w$ 이므로 물건 2를 배낭에 담을 수 없다. 따라서 $K[2,1] = K[2,2] = K[2,3] = 0$ 이다.

■ 배낭 용량 $w = 4kg$ 일 때, 물건 2($w_2 = 4kg$)을 배낭에 담으려 한다. 이번에는 $w_2 = w$ 이므로 물건 2를 배낭에 담을 수 있다. 따라서

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

이다.

- 배낭 용량 $w = 5kg$ 일 때, 물건 2($w_2 = 4kg$)을 배낭에 담으려 한다. 이번에도 $w_2 = w$ 이므로 물건 2를 배낭에 담을 수 있다. 따라서

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

이다.

- 참고로, 이번에는 물건 1($w_1 = 5kg$)도 배낭에 담을 수 있으므로 물건1을 담을 때와 물건 2를 담을 때의 최대 가치를 비교하여 더 큰 가치를 얻는 물건 2를 배낭에 담은 것이다.

- 배낭 용량 $w = 6, 7, 8kg$ 일 때도 각각의 경우 물건 1을 빼내고 물건 2를 배낭에 담는 것이 더 큰 가치를 얻는다.
따라서 $K[2,6] = K[2,7] = K[2,8] = 40$ 이다.

- 배낭 용량 $w = 9kg$ 일 때, 물건 2($w_2 = 4kg$)을 배낭에 담으려 한다. 이번에는 $w_2 < w$ 이므로 물건 2를 배낭에 담을 수 있다. 따라서

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

- 참고로, 이번 경우에는 물건 2를 담고도 물건 1을 담을 수 있어서 두 개의 물건을 모두 배낭에 담을 수 있었다.

■ 배낭 용량 $w = 10kg$ 일 때도 물건 1과 물건 2를 모두 배낭에 담을 수 있으므로 $K[2,10] = 50$ 이다.

● 다음은 물건 1과 2에 대해서만 배낭의 용량을 1에서 C까지 늘려가며 알고리즘을 수행한 결과이다.

배낭용량 w (C=10kg)			0	1	2	3	4	5	6	7	8	9	10
물건 i	가치 v_i	무게 w_i	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										
4	50	3	0										

- 나머지 물건 3과 4에 대해서도 배낭의 용량을 1에서 C까지 늘려가며 알고리즘을 수행한 결과는 다음과 같다.

배낭용량 w $(C=10kg)$			0	1	2	3	4	5	6	7	8	9	10
물건 i	가치 v_i	무게 w_i	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

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

시간복잡도

- 하나의 부분 문제에 대한 해를 구할 때의 시간복잡도는 line 5에서의 무게를 한 번 비교한 후, line 6에서는 1개의 부분 문제의 해를 참조하고, line 8에서는 2개의 해를 참조해 계산하므로 $O(1)$이 걸린다.

- 그런데 부분 문제의 수는 배열 K의 원소 수인 $n \times C$개이다. 여기서 C는 배낭의 용량이다.

- 따라서 Knapsack 알고리즘의 시간복잡도는 $O(1) \times n \times C = O(nC)$ 이다.

응용 사례

- 배낭 문제는 다양한 분야에서 의사 결정 과정에 활용된다. 예를 들어, 원자재의 버리는 부분을 최소화시키기 위한 자르기 문제, 금융 포트폴리오와 자산 투자의 선택, 암호 생성 시스템 등에 활용된다.

6.5 동전 거스름돈 문제

- 잔돈을 동전으로 거슬러 받아야 할 때
  누구나 적은 수의 동전으로 거스름돈을 받고 싶어 한다.
  - 대부분의 경우 그리디 알고리즘으로 해결되나
    해결 못하는 경우도 있다.
  - 동적 계획 알고리즘은 모든 동전 거스름돈 문제에 대하여
    항상 최적해를 찾는다.

문제 해결 아이디어

- 동전 거스름돈 문제에 주어진 문제 요소들을 생각해보자.
  - 정해진 동전의 종류 $d_1, d_2, \cdots, d_k$가 있고, 거스름돈 n원이 있다. 단, $d_1 > d_2 > \cdots > d_k = 1$ 이라고 하자.
  - 예를 들어, 우리나라의 동전 종류는 5개로서 $d_1 = 500,\ d_2 = 100,\ d_3 = 50,\ d_4 = 10,\ d_5 = 1$ 이다.
  - 앞에서 공부한 배낭 문제의 동적 계획 알고리즘을 살펴보면, 배낭의 용량을 1kg씩 증가시켜 문제를 해결한다. 여기서 힌트를 얻어, 동전 거스름돈 문제도 1원씩 증가시켜 문제를 해결할 수 있다.

- 즉, 거스름돈을 배낭의 용량, 동전을 물건으로 생각하여 부분 문제들의 해를 아래와 같이 1차원 배열 C에 저장하자.

1원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[1]$

2원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[2]$

:

j원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[j]$

:

n원을 거슬러 받을 때 사용되는 최소의 동전 수 $C[n]$

동전 거스름돈 문제의 부분 문제

- 구체적으로 $C[j]$를 구하는데 어떤 부분 문제가 필요할까?
  - j원을 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 j원에 필요하면 $(j - 500)$원의 해, 즉, $C[j - 500] = C[j - d_1]$에 500원짜리 동전 1개를 추가한다.
    - 100원짜리 동전이 거스름돈 j원에 필요하면 $(j - 100)$원의 해, 즉, $C[j - 100] = C[j - d_2]$에 100원짜리 동전 1개를 추가한다.
    - 50원짜리 동전이 거스름돈 j원에 필요하면 $(j - 50)$원의 해, 즉, $C[j - 50] = C[j - d_3]$에 50원짜리 동전 1개를 추가한다.
    - 10원짜리 동전이 거스름돈 j원에 필요하면 $(j - 10)$원의 해, 즉, $C[j - 10] = C[j - d_4]$에 10원짜리 동전 1개를 추가한다.
    - 1원짜리 동전이 거스름돈 j원에 필요하면 $(j - 1)$원의 해, 즉, $C[j - 1] = C[j - d_5]$에 1원짜리 동전 1개를 추가한다.

- 앞의 5가지 중에서 당연히 가장 작은 값을 $C[j]$로 정해야 한다. 따라서 $C[j]$는 아래와 같이 정의된다.

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

  - 즉, $i = 1,2, \ldots, k$ 까지 변하면서 $d_1, d_2, \cdots, d_k$ 각각에 대하여 해당 동전을 거스름돈에 포함시킬 경우의 동전 수를 고려하여 최소값을 $C[j]$로 정한다.

- 따라서, 부분 문제간의 함축적 순서는 다음과 같다.

$$C[j - d_i] \longrightarrow C[j]$$

DPCoinChange 알고리즘

$\boldsymbol{DPCoinChange}$

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 거스름돈 액수
4. $\quad for\ i\ =\ 1\ to\ k\ \{$ // 각각의 동전 $d_1 \sim d_k$에 대해
   // $C[j] = min_{1 \le i \le k}\{C[j - d_i] + 1\},\ if\ j \ge d_i$
5. $\quad\quad if\ (d_i \le\ j)\ and\ (C[j - d_i] + 1 < C[j])$
6. $\quad\quad\quad C[j] = C[j - d_i] + 1$

   $\quad \}$

   $\}$

7. $return\ C[n]$

- Line 1: 배열 C의 모든 원소를 ∞로 초기화 한다. ∞ 로 초기화하는 이유는 문제에서 거슬러 받는 동전의 최소 수를 구하기 때문이다.
- Line 2: $C[0] = 0$으로 초기화한다. 즉, 거스름돈 0원을 거슬러 주기 위해서는 0개의 동전이 필요하다는 의미이다. 이는 나중에 line 5의 $C[j - d_i]$를 참조하는 부분에서 $C[0]$이 필요한 경우를 위해서이다.
- Line 3~6: for루프에서는 거스름돈 액수 j를 1원부터 1원씩 증가시키며, line 4~6에서 $min_{1 \le i \le k}\{C[j - d_i] + 1\}$을 $C[j]$로 정한다.
- Line 4~6: 가장 큰 액면의 동전 d_1부터 1원짜리 동전 d_k까지 차례로 동전을 고려해보고, 그 중 가장 적은 동전 수를 $C[j]$로 결정한다. 단, 거스름돈 액수 j원보다 크지 않은 동전에 대해서만 고려한다.

DPCoinChange 알고리즘 수행과정

- 동전의 종류 $d_1 = 16, d_2 = 10, d_3 = 5, d_4 = 1$이고,
  거스름돈 $n = 20$일 때 알고리즘의 수행되는 과정이다.

- Line 1~2에서 배열 C를 아래와 같이 초기화시킨다.

j	0	1	2	3	4	5	⋯	16	17	18	19	20
C	0	∞	∞	∞	∞	∞	⋯	∞	∞	∞	∞	∞

- 거스름돈 $j = 1 \sim 4$원일 때
  - $i = 1 \sim 3$인 경우(16원, 10원, 5원 동전), $(d_i \leq j)$가 '거짓'으로 동전의 크기가 거스름돈 크기보다 크므로 고려대상에서 제외된다.
  - $i = 4$일 때, $d_4 = 1$원짜리 동전을 고려하여 각 j에서 1을 뺀 $(j-1)$의 해를 사용하여 $C[j] = C[j-1] + 1$이 된다.

$$C[1] = C[j-1] + 1 = C[1-1] + 1 = C[0] + 1 = 0 + 1 = 1$$
$$C[2] = C[j-1] + 1 = C[2-1] + 1 = C[1] + 1 = 1 + 1 = 2$$
$$C[3] = C[j-1] + 1 = C[3-1] + 1 = C[2] + 1 = 2 + 1 = 3$$
$$C[4] = C[j-1] + 1 = C[4-1] + 1 = C[3] + 1 = 3 + 1 = 4$$

j	0	1	2	3	4	⋯
C	0	1	2	3	4	⋯

- 거스름돈 $j = 5$원일 때
  - $i = 1 \sim 2$인 경우(16원, 10원 동전), $(d_i \leq j)$가 '거짓'으로 고려대상에서 제외
  - $i = 3$일 때, $d_3 = 5$원짜리 동전에 대해서 $(C[j - d_3] + 1 < C[j]) = (C[5 - 5] + 1 < C[5]) = (C[0] + 1 < \infty) = (0 + 1 < \infty)$이 '참'이 되어 $C[j] \; = \; C[j - d_i] + 1$이 수행되어 $C[5] = 1$이 된다.
  - $i = 4$일 때, $d_4 = 1$원짜리 동전에 대해서 $(C[j - d_4] + 1 < C[j]) = (C[5 - 1] + 1 < C[5]) = (C[4] + 1 < 1) = (4 + 1 < 1)$이 '거짓'이 되어 $C[5] = 1$을 유지한다.
  - 즉, 5원짜리 동전 1개를 거슬러 주면 된다.

j	0	1	2	3	4	5	…
C	0	1	2	3	4	1	…

- 거스름돈 $j = 6$원일 때
  - $i = 3$일 때, $d_3 = 5$원짜리 동전에 대해서 $(C[j - d_3] + 1 < C[j]) = (C[6-5] + 1 < C[6]) = (C[1] + 1 < \infty) = (1 + 1 < \infty)$이 '참'이 되어 $C[j] = C[j - d_i] + 1$이 수행되어 $C[6] = 2$이 된다.

  - $i = 4$일 때, $d_4 = 1$원짜리 동전에 대해서 $(C[j - d_4] + 1 < C[j]) = (C[6-1] + 1 < C[6]) = (C[5] + 1 < 2) = (1 + 1 < 2)$는 '거짓'이 되어 $C[j]$는 변경되지 않는다. 사실 $i = 3$일 때와 같은 값이 나온다.

j	0	1	2	3	4	5	6	⋯
C	0	1	2	3	4	1	2	⋯

- 거스름돈 $j = 7 \sim 9$원일 때는 $j = 6$원일 때와 마찬가지로
  - $i = 3$일 때, $d_3 = 5$원짜리 동전에 대해서 다음과 같이 수행된다.

    $C[7] = C[j-5] + 1 = C[7-5] + 1 = C[2] + 1 = 2 + 1 = 3$

    $C[8] = C[j-5] + 1 = C[8-5] + 1 = C[3] + 1 = 3 + 1 = 4$

    $C[9] = C[j-5] + 1 = C[9-5] + 1 = C[4] + 1 = 4 + 1 = 5$

  - $i = 4$일 때, $d_4 = 1$원짜리 동전에 대해서 다음과 같이 수행되어 $C[j]$는 변경되지 않는다. (모두 $i = 3$일 때와 같은 값이 나온다.)

    $C[j-1] + 1 = C[7-1] + 1 = C[6] + 1 = 2 + 1 = 3$

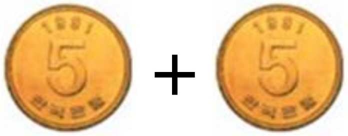
    $C[j-1] + 1 = C[8-1] + 1 = C[7] + 1 = 3 + 1 = 4$

    $C[j-1] + 1 = C[9-1] + 1 = C[8] + 1 = 4 + 1 = 5$

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

- 거스름돈 $j = 10$원일 때
  - $i = 2$일 때, $d_2 = 10$원짜리 동전에 대해서 $(C[j - d_2] + 1 < C[j]) = (C[10 - 10] + 1 < C[10]) = (C[0] + 1 < \infty) = (0 + 1 < \infty)$이 '참'이 되어 $C[j] = C[j - d_i] + 1$이 수행되어 $C[10] = 1$이 된다.

  - $i = 3$일 때, $d_3 = 5$원짜리 동전에 대해서 $(C[j - d_3] + 1 < C[j]) = (C[10 - 5] + 1 < C[10]) = (C[5] + 1 < 1) = (1 + 1 < \infty)$이 '거짓'이 되어 $C[j]$는 변경되지 않는다.



- $i = 4$일 때, $d_4 = 1$원짜리 동전에 대해서 $(C[j - d_4] + 1 < C[j]) = (C[10 - 1] + 1 < C[10]) = (C[9] + 1 < 1) = (5 + 1 < 1\ )$이 '거짓'이 되어 $C[j]$는 변경되지 않는다.

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

- 거스름돈 $j = 11\sim19$원일 때 다음과 같은 결과가 나온다.
  - $C[15] = 2$

  - $C[16] = 1$

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										∞

- 거스름돈 $j = 20$원일 때
  - $i = 1$일 때, $d_1 = 16$원짜리 동전에 대해서 $(C[j - d_1] + 1 < C[j]) = (C[20 - 16] + 1 < C[20]) = (C[4] + 1 < \infty) = (4 + 1 < \infty)$이 '참'이 되어 $C[j] \ = \ C[j - d_i] + 1$이 수행되어 $C[20] = 5$이 된다.

  - $i = 2$일 때, $d_2 = 10$원짜리 동전에 대해서 $(C[j - d_2] + 1 < C[j]) = (C[20 - 10] + 1 < C[20]) = (C[10] + 1 < 5) = (1 + 1 < 5)$이 '참'이 되어 $C[j] \ = \ C[j - d_i] + 1$이 수행되어 $C[20] = 2$가 된다.

- $i = 3$일 때, $d_3 = 5$원짜리 동전에 대해서 $(C[j - d_3] + 1 < C[j]) = (C[20 - 5] + 1 < C[20]) = (C[15] + 1 < 2) = (2 + 1 < 2)$이 '거짓'이 되어 $C[j]$는 변경되지 않는다.

- $i = 4$일 때, $d_4 = 1$원짜리 동전에 대해서 $(C[j - d_4] + 1 < C[j]) = (C[20 - 1] + 1 < C[20]) = (C[19] + 1 < 2) = (4 + 1 < 2)$이 '거짓'이 되어 $C[j]$는 변경되지 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

- 결국, 거스름돈 20원에 대한 최종해는 $C[20] = 2$개의 동전이다.
- 그리디 알고리즘의 경우,
20원에 대해 16원짜리 동전을 먼저 '욕심 내어' 취하고,
남은 4원에 대해 1원짜리 4개를 취하여
모두 5개의 동전이 해라고 답하였다.

그리디 알고리즘의 해

동적 계획 알고리즘의 해

시간복잡도

- DPCoinChange 알고리즘의 시간복잡도는 $O(nk)$ 이다.
  이는 거스름돈 j가 $1 \sim n$원까지 변하며, 각각의 j에 대해서 최악의 경우 k개의 동전을 1번씩 고려하기 때문이다.

요약

- 동적 계획(Dynamic Programming) 알고리즘은 최적화 문제를 해결하는 알고리즘으로서, 입력 크기가 작은 '부분 문제'들을 모두 해결한 후에 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여, 최종적으로 원래 주어진 입력의 문제를 해결하는 알고리즘이다.
- 동적 계획 알고리즘에는 부분 문제들 사이에 '의존적 관계'가 존재한다.
- 동적 계획 알고리즘은 부분 문제들 사이의 '관계'를 빠짐없이 고려하여 문제를 해결한다.
- 동적 계획 알고리즘은 최적 부분 구조(optimal substructure) 또는 최적성 원칙(principle of optimality) 특성을 가지고 있다.

Q&A