Chapter 5-2

Dynamic Programming

Edit Distance 문제

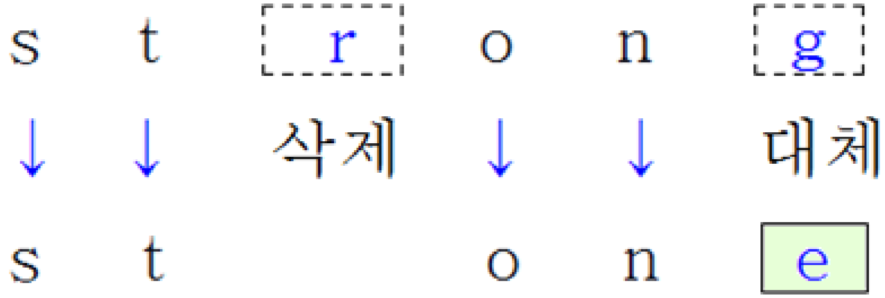
- Edit Distance (편집 거리)
  - 삽입 (insert), 삭제 (delete), 대체 (substitute) 연산을 사용하여 string (문자열), S를 수정하여 다른 string, T로 변환하고자 할 때 필요한 최소의 편집 연산 횟수
  - ‘strong’ => ‘stone’

s	t		r	o	n	g
↓	↓	삽입	삭제	삭제	↓	대체
s	t	o			n	e

  - 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

Edit Distance 문제

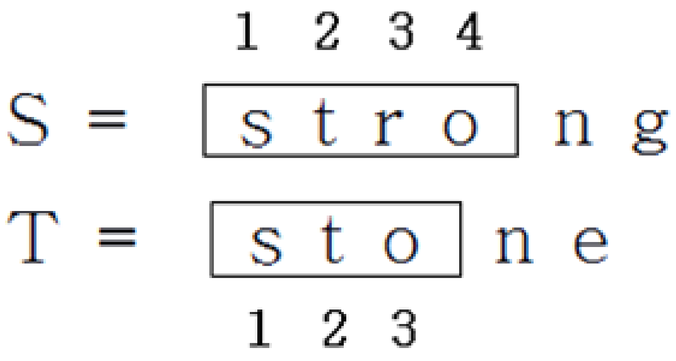
- 다른 시도
  - ‘s’와 ‘t’는 그대로 사용한 후, ‘r’을 삭제하고, ‘o’와 ‘n’을 그대로 사용한 후, ‘g’를 ‘e’로 대체하여, 총 2회의 편집 연산만 수행되고, 이는 최소 편집 횟수임



  - S를 T로 바꾸는데 어떤 연산을 어느 문자에 수행하는가에 따라서 편집 연산 횟수가 달라짐

Edit Distance 문제

- 부분 문제들의 표현 방법
  - 접두부 (prefix)를 고려
  - 'strong' => 'stone'에 대해
  - 예를 들어, 'stro'를 'sto'로 바꾸는 편집 거리를 미리 알면, 'ng'를 'ne'로 바꾸는 편집 거리를 찾음으로써 주어진 입력에 대한 편집 거리를 구할 수 있음



Edit Distance 문제

- 부분 문제 정의
  - $|S| = m$, $|T| = n$
    - $S = s_1\ s_2\ s_3\ s_4\ \dots\ s_m$
    - $T = t_1\ t_2\ t_3\ t_4\ \dots\ t_n$
  - $E[i, j]$ = 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] 임

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

Edit Distance 문제

- 'strong' => 'stone'에 대해
  - s_1 -> t_1 ['s' => 's']: s_1 = t_1 = 's'이므로 E[1, 1] = 0
  - s_1 -> t_1t_2 ['s' => 'st']: s_1 = t_1 = 's'이고, 't'를 삽입하므로 E[1, 2] = 1
    - E[1, 2] = E[1, 1] + 1
  - s_1s_2 -> t_1 ['st' => 's']: s_1 = t_1 = 's'이고, 't'를 삭제하여 E[2, 1] = 1
    - E[2, 1] = E[1, 1] + 1
  - s_1s_2 -> t_1t_2 ['st' => '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_2=t_2='t'이므로 E[2, 2] = E[1, 1]+0 = 0

Edit Distance 문제

- E[4, 3]의 계산 방법
  - $s_1s_2s_3s_4$ -> $t_1t_2t_3$ ['stro' => 'sto']

T

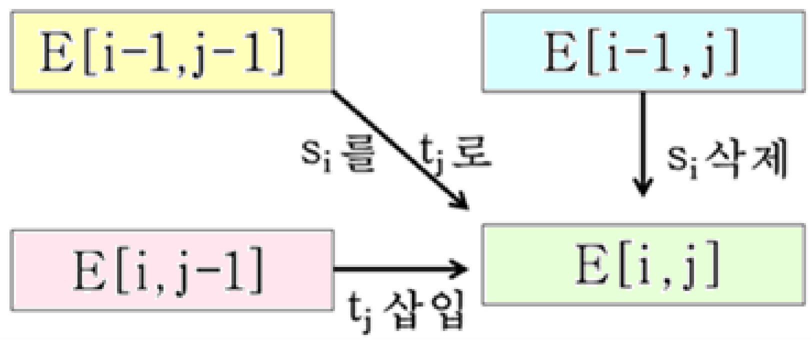
E		ε	s	t	o
	i \ j	0	1	2	3
ε	0	0	1	2	3
s	1	1	0	1	2
t	2	2	1	0	1
r	3	3	2	1	1
o	4	4	3	2	1

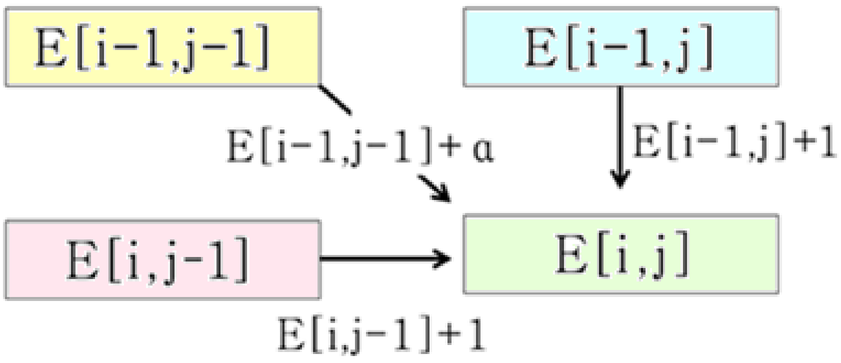
S

  - E[4, 2]의 해를 알면, t_3 = 'o'를 삽입하면 E[4, 2] + 1
  - E[3, 3]의 해를 알면, s_4 = 'o'를 삭제하면 E[3, 3] + 1
  - E[3, 2]의 해를 알면, s_4 = 'o'과 t_3='o'이 같으므로 E[3, 2] + 0

Edit Distance 문제

- E[i, j]의 계산 방법





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

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

Edit Distance 문제

- 초기화

T

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

- 0번 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 삽입 연산 횟수를 각각 나타냄
- 0번 열이 0, 1, 2, …, m으로 초기화된 의미
  - String, T를 ε로 만들기 위해서, S의 문자를 위에서 아래로 하나씩 없애는데 필요한 삭제 연산 횟수를 각각 나타냄

Edit Distance 문제

- 알고리즘

EditDistance

입력: string, S, T, 단, S와 T의 길이는 각각 m과 n

출력: S를 T로 변환하는 편집 거리, E[m, n]

```
for i=0 to m    E[i, 0] = j    // 0번 열의 초기화
for j=0 to n    E[0, j] = j    // 0번 행의 초기화
for i=1 to m
    for j=1 to n
        E[i, j] = min{E[i, j-1]+1, E[i-1, j]+1, E[i-1, j-1]+α}
return E[m, n]
```

Edit Distance 문제

- 알고리즘 수행과정
  - ‘strong’을 ‘stone’으로 바꾸는데 필요한 편집 거리

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**

Edit Distance 문제

- 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	ε	s
S	i \ j	0	1
ε	0	0	1
s	1	1	0

Edit Distance 문제

- E[2, 2]
  - E[2, 2] = min{E[2, 1]+1, E[1, 2]+1, E[1, 1]+α}

    = min{(1+1), (1+1), (0+0)}

    = 0

	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

Edit Distance 문제

- E[3, 2]
  - E[3, 2] = min{E[3, 1]+1, E[2, 2]+1, E[2, 1]+α}

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

    = 1

	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

Edit Distance 문제

- 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	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

Edit Distance 문제

- E[5, 4]
  - E[5, 4] = min{E[5, 3]+1, E[4, 4]+1, E[4, 3]+α}

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

    = 1

	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

Edit Distance 문제

- E[6, 5]
  - E[6, 5] = min{E[6, 4]+1, E[5, 5]+1, E[5, 4]+α}

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

    = 2

	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

Edit Distance 문제

- Time Complexity
  - EditDistance 알고리즘의 시간 복잡도는 O(mn)
    - m, n은 두 string의 각각의 길이
  - 총 부분 문제의 수가 배열 E의 원소 수인 m X n이고, 각 부분 문제 (원소)를 계산하기 위해서 주위의 3개의 부분 문제들의 해 (원소)를 참조한 후 최소값을 찾는 것이므로 O(1) 시간 소요

Knapsack 문제

- 배낭 문제
  - n개의 물건과 각 물건 i의 무게 w_i와 가치 v_i가 주어지고, 배낭의 용량이 C일 때, 배낭에 담을 수 있는 물건의 가치는?
  - 단, 배낭에 담은 물건의 무게의 합이 C를 초과하지 말아야 하고, 각 물건은 1개씩만 있음
  - 이러한 배낭 문제를 0-1 배낭 문제라고 정의함

Knapsack 문제

- 부분 문제
  - 문제에는 물건, 물건의 무게, 물건의 가치, 배낭의 용량, 모두 4가지의 요소가 있음
  - 물건과 물건의 무게는 부분 문제를 정의하는 데에 필요
  - 문제의 정의

    K[i, w] = 물건 1~i 까지만 고려하고, (임시) 배낭의 용량이 w일 때의 최대 가치

    단, i = 1, 2, ..., n 이고, w = 1, 2, 3, ..., C

    - 문제의 최적해 = K[n, C]

Knapsack 문제

- 알고리즘
  - Knapsack

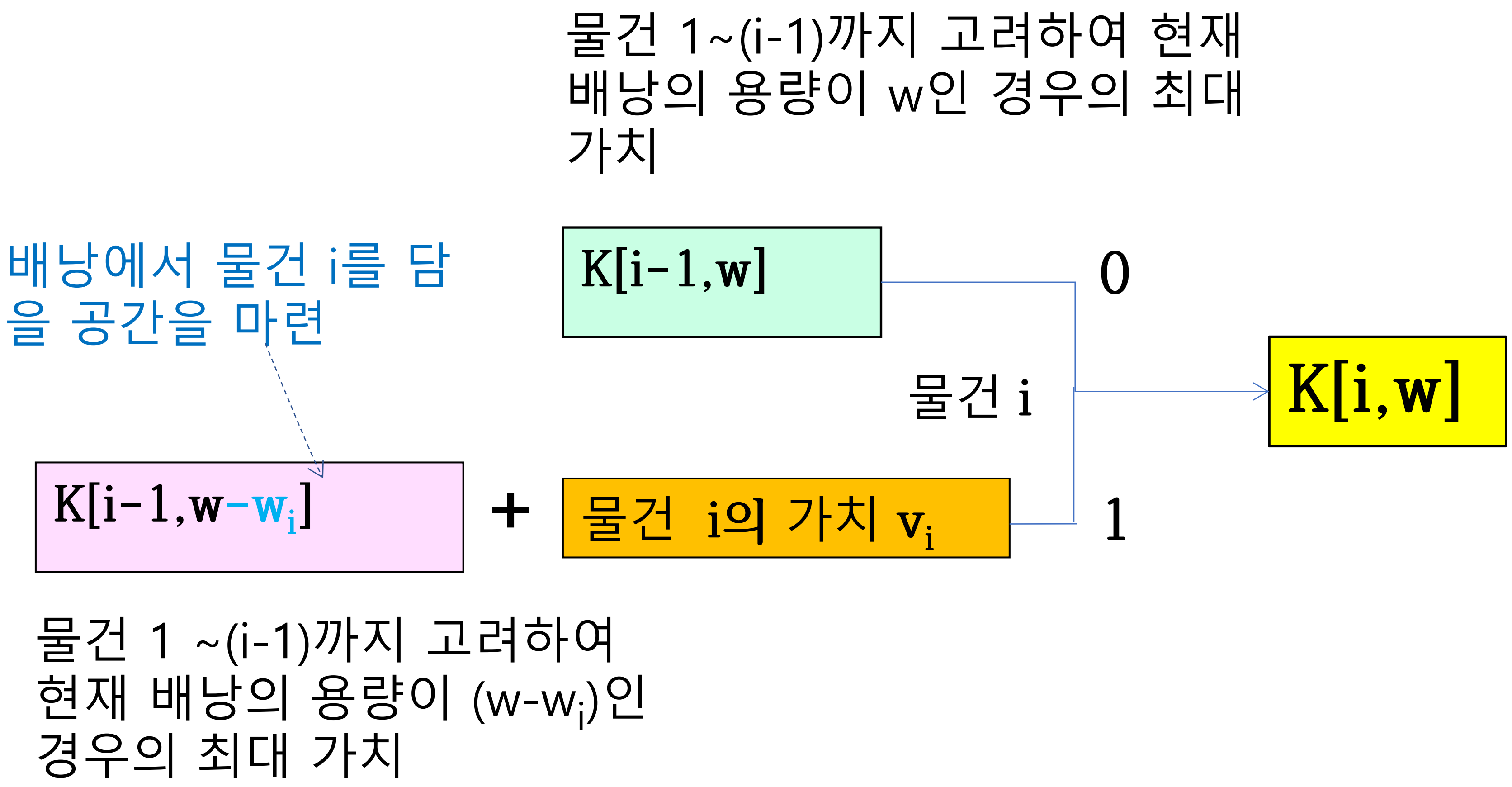
입력: 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 w_i와 가치 v_i, (단, i = 1, 2, …, n)

출력: K[n, C]

```
for i = 0 to n   K[i, 0] = 0          // 배낭의 용량이 0일 때
for w = 0 to C   K[0, w] = 0          // 물건 0이란 어떤 물건도 고려하지 않을 때
for i = 1 to n
    for w = 1 to C                    // w는 배낭의 (임시) 용량
        if (w_i > w)                  // 물건 i의 무게가 임시 배낭 용량을 초과하면
            K[i, w] = K[i-1, w]
        else                          // 물건 i를 배낭에 담지 않을 경우와 담을 경우를 고려
            K[i, w] = max{ K[i-1, w], K[i-1, w-w_i]+v_i }
return K[n, C]
```

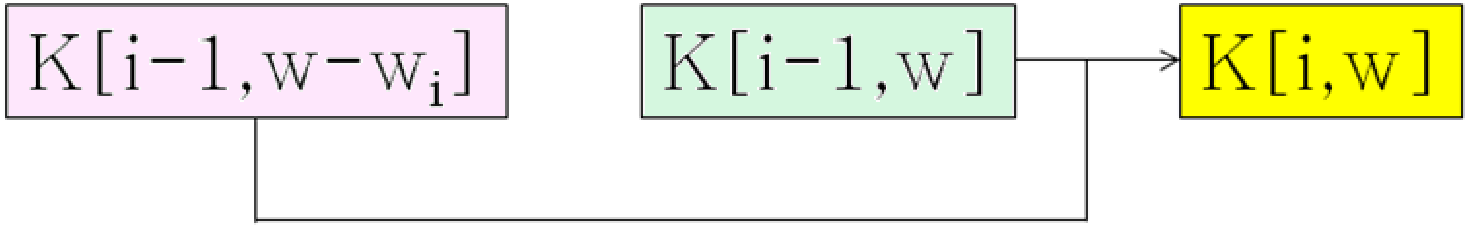
Knapsack 문제

- 알고리즘



Knapsack 문제

- 알고리즘
  - 부분 문제 간의 함축적 순서



Knapsack 문제

- 알고리즘 수행 과정
  - 배낭의 용량 C = 10 kg

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

Knapsack 문제

- 알고리즘 수행 과정
  - 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										

Knapsack 문제

- 알고리즘 수행 과정
  - w = 1, 2, 3, 4일 때, 각각 물건 1을 담을 수 없음
    - K[1, 1] = 0, K[1, 2] = 0, K[1, 3] = 0, K[1, 4] = 0

Knapsack 문제

- 알고리즘 수행 과정
  - W = 5 (배낭의 용량이 5 kg)일 때,
    - K[1, 5] = 10

5kg

10만원

Knapsack 문제

- 알고리즘 수행 과정
  - w = 6, 7, 8, 9, 10일 때,
    - K[1, 6] = K[1, 7] = K[1, 8] = K[1, 9] = K[1, 10] = 10

Knapsack 문제

- 알고리즘 수행 과정
  - 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										

Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 1, 2, 3 (배낭의 용량이 각각 1, 2, 3 kg)일 때
      - K[2, 1] = 0, K[2, 2] = 0, K[2, 3] = 0

Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 4 (배낭의 용량이 4 kg)일 때
      - K[2, 4] = 40

Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 5 (배낭의 용량이 5 kg)일 때
      - $K[2, 5] = \max\{K[i-1, w], K[i-1, w-w_i]+v_i\}$

        $= \max\{K[2-1, 5], K[2-1, 5-4]+40\}$

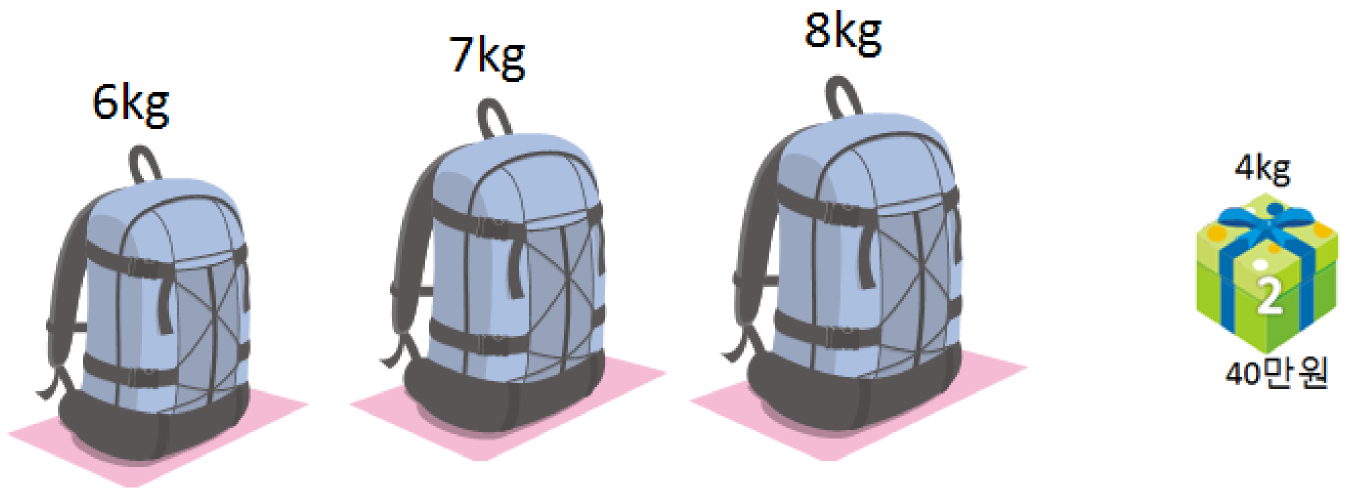
        $= \max\{K[1, 5], K[1, 1]+40\}$

        $= \max\{10, 0+40\}$

        $= \max\{10, 40\} = 40$
      - 물건 1을 배낭에서 빼낸 후, 물건 2를 담음
      - 그 때의 가치는 40

Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 6, 7, 8일 때
      - 각각의 경우도 물건 1을 빼내고 물건 2를 배낭에 담는 것이 더 큰 가치를 얻음
      - K[2, 6] = K[2, 7] = K[2, 8] = 40



Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 9 (배낭의 용량이 9 kg)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 있음

Knapsack 문제

- 알고리즘 수행 과정
  - 물건 2를 고려했을 때
    - w = 10 (배낭의 용량이 10 kg)일 때
      - 물건 1, 2를 배낭에 둘 다 담을 수 있음

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	0										

Knapsack 문제

- 알고리즘 수행 과정

C

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	$\boxed{90}$

- 최적해 K[4, 10] = 90

Knapsack 문제

- Time Complexity
  - 1개의 부분 문제에 대한 해를 구할 때의 time complexity
    - Line 5에서의 무게를 1번 비교한 후 line 6에서는 1개의 부분 문제의 해를 참조하고 line 8에서는 2개의 해를 참조한 계산이므로 O(1) 시간
    - 부분 문제의 수는 배열 K의 원소의 개수인 n X C
      - C = 배낭의 용량
    - Knapsack 알고리즘의 time complexity
      - O(1) X n X C = O(nC)

Coin Change 문제

- 동전 거스름돈 문제
  - 대부분의 경우 Greedy algorithm으로 해결되나, 해결 못하는 경우도 있음
  - DP 알고리즘은 모든 동전 거스름돈 문제에 대해 항상 최적해를 가짐

Coin Change 문제

- 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

Coin Change 문제

- C[j]를 계산하는 데에 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

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

Coin Change 문제

- 알고리즘
  - DPCoinChange

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

    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 액수
4.   for i = 1 to k // 액면이 가장 높은 동전부터 1원짜리 동전까지
5.   if (d_i <= j) and (C[j-d_i]+1 < C[j])
6.   C[j] = C[j-d_i] + 1
7. return C[n]

Coin Change 문제

- 알고리즘 수행 과정
  - d_1 = 16, d_2 = 10, d_3 = 5, d_4 = 1이고, 거스름돈 n=20일 때

Coin Change 문제

- 알고리즘 수행 과정
  - Line 1~2: 배열 C 초기화

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

Coin Change 문제

- 알고리즘 수행 과정
  - 거스름돈 1원~4원까지
    - C[1] = C[j-1]+1 = C[1-1]+1 = C[0]+1 = 0+1 = 1

j	0	1
	0	∞

⇨

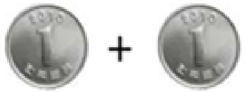
j	0	1
	0	1

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

j	1	2
	1	∞

⇨

j	1	2
	1	2



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

j	2	3
	2	∞

⇨

j	2	3
	2	3

    - C[4] = C[j-1]+1 = C[4-1]+1 = C[3]+1 = 3+1 = 4

j	3	4
	3	∞

⇨

j	3	4
	3	4

Coin Change 문제

- 알고리즘 수행 과정
  - 거스름돈 5원

1개

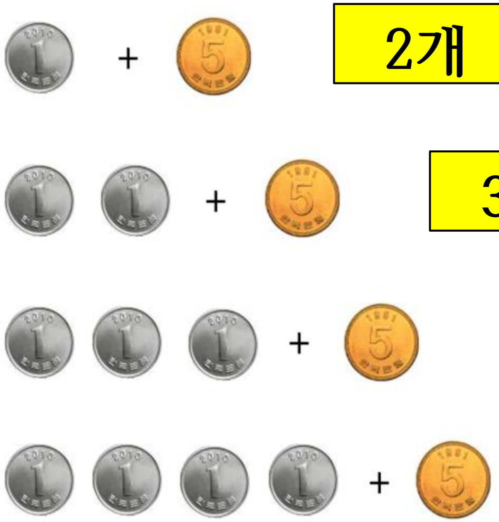
j	0	1	2	3	4	5
	0	1	2	3	4	∞

5개

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

Coin Change 문제

- 알고리즘 수행 과정
  - 거스름돈 6, 7, 8, 9원



2개

3개

4개

5개

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

Coin Change 문제

- 알고리즘 수행 과정
  - 거스름돈 10원

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

1개

Coin Change 문제

- 알고리즘 수행 과정
  - 거스름돈 20원

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

2개

11	12	13	14	15	16	17	18	19	20
2	3	4	5	2	1	2	3	4	2

Coin Change 문제

- 알고리즘 수행 결과
  - 거스름돈 20원에 대한 최종해 = C[20] = 2
  - Greedy algorithm은 20원에 대해 16원 동전을 먼저 '욕심내어 ' 취하고, 4원이 남게 되어, 1원 4개를 취하여, 모두 5개 동전의 해를 구함

그리디 알고리즘의 해

동적 계획 알고리즘의 해

Coin Change 문제

- Time Complexity
  - O(nk)
  - 거스름돈 j가 1원~n원까지 변하며, 각각의 j에 대해서 최대 모든 동전 (d_1, d_2, …, d_k)을 (즉, k개를) 1번씩 고려하기 때문

Summary

- 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 algorithm에는 부분 문제들 사이에 함축적 순서가 존재함

Summary

- All Pairs Shortest Paths 문제를 위한 Floyd-Warshall 알고리즘은 $O(n^3)$의 time complexity를 가짐
  - 경유 가능한 점들을 점 1로부터 시작하여, 점 1과 2, 그 다음엔 점 1, 2, 3으로 하나씩 추가하여, 마지막에는 점 1에서 점 n까지의 모든 점을 경유 가능한 점들로 고려하면서, 모든 쌍의 최단 경로의 거리를 계산
- Chained Matrix Multiplications 문제를 위한 $O(n^3)$ DP algorithm은 이웃하는 행렬들끼리 곱하는 모든 부분 문제들을 해결하여 최적해를 찾음

Summary

- 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 계산함. Time Complexity는 O(mn).
- Knapsack 문제를 위한 DP 알고리즘은 부분 문제 K[i, w]를 물건 1~i까지만 고려하고, (임시) 배낭의 용량이 w일 때의 최대 가치로 정의하며, i를 1~물건 수인 n까지, w를 1~배낭 용량 C까지 변화시키며 해를 찾음. Time Complexity는 O(nC)

Summary

- Coin Change 문제는 1원씩 증가시켜 문제를 해결함. Time Complexity는 O(nk)
- DP 알고리즘은 부분 문제들 사이의 관계를 빠짐없이 고려하여 문제를 해결
- DP 알고리즘은 최적 부분 구조 (optimal substructure) 또는 최적성 원칙 (principle of optimality) 특성을 가짐
    - 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있음
    - Greedy algorithm도 같은 속성을 가짐