5. 동적 계획 알고리즘2
(Dynamic Programming)

Prof. Jongmin Lee

Wonkwang University

동적 계획 알고리즘

- **동적 계획 (Dynamic Programming) 알고리즘**
  - 최적화 문제를 해결하는 알고리즘

- **동적 계획 알고리즘의 문제 해결 방법**
  - 입력 크기가 작은 부분 문제들을 모두 해결한 후에
  - 그 해들을 이용하여 보다 큰 크기의 부분 문제들을 해결하여
  - 최종적으로 원래 주어진 입력의 문제를 해결

- 0/1배낭문제
- 동전거스름돈문제

4. 배낭 문제

- 배낭 (Knapsack) 문제
  - n개의 물건과 각 물건 i의 무게 w_i와 가치 v_i가 주어지고, 배낭의 용량은 C일 때, 배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제이다.
  - 단, 배낭에 담은 물건의 무게의 합이 C를 초과하지 말아야 하고, 각 물건은 1개씩만 있다.
  - 이러한 배낭 문제를 0-1 배낭 문제라고 한다.
    - 각 물건이 배낭에 담기지 않은 경우는 0, 담긴 경우는 1로 여기기 때문

•

배낭 문제

- 문제의 주어진 조건을 살펴보면 물건, 물건의 무게, 물건의 가치, 배낭의 용량, 모두 4가지의 요소가 있다.

- 물건과 물건의 무게는 부분문제를 정의하는데 필요하다.

- 또한 물건을 배낭에 담으려고 할 경우에 배낭 용량의 초과 여부를 검사해야 한다.

배낭 문제

- 따라서 배낭 문제의 부분문제를 아래와 같이 정의할 수 있다.

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

  단, i = 1, 2, ⋯, n이고, w = 1, 2, 3, ⋯, C이다.

  ○ 그러므로 문제의 최적해는 K[n,C]이다.

  ○ 여기서 주의하여 볼 것은 배낭의 용량이 C이지만, 배낭의 용량을 1부터 C까지 1씩 증가시킨다는 것이다.

  ○ 이 때문에 C의 값이 매우 크면, 알고리즘의 수행시간 너무 길어지게 된다.

  ○ 따라서 다음의 알고리즘은 제한적인 입력에 대해서만 효용성을 가진다.

Knapsack 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]

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_i > w)   // 물건 i의 무게가 임시 배낭 용량을 초과하면
6.   K[i,w] = K[i-1,w]
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]

Knapsack 알고리즘

- Line 1
  - 2차원 배열 K의 0번 열을 0으로 초기화시킨다.
  - 그 의미는 배낭의 (임시) 용량이 0일 때, 물건 1~n까지 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 C까지 각각 증가시켜도 담을 물건이 없으므로 각각의 최대 가치는 0이다.
- Line 3~8
  - 물건을 1에서 n까지 하나씩 고려하여 배낭의 (임시) 용량을 1에서 C까지 각각 증가시키며, 다음을 수행한다.

Knapsack 알고리즘

- Line 5~6
  - 현재 배낭에 담아보려고 고려하는 물건 i의 무게 w_i가 (임시) 배낭 용량 w보다 크면 물건 i를 배낭에 담을 수 없으므로, 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
  - 만일 현재 고려하는 물건 i의 무게 w_i가 현재 배낭의 용량 w보다 같거나 작으면, 물건 i를 배낭에 담을 수 있다.
  - 그러나 현재 상태에서 물건 i를 추가로 배낭에 담으면 배낭의 무게가 $(w+w_i)$로 늘어난다.
  - 따라서 현재의 배낭 용량인 w를 초과하게 되어, 물건 i를 추가로 담을 수는 없다.

Knapsack 알고리즘

- 그러므로 물건 i를 배낭에 담기 위해서는 2가지 경우를 살펴보아야 한다.
  - 물건 i를 배낭에 담지 않는 경우, K[i,w] = K[i-1,w]가 된다.
  - 물건 i를 배낭에 담는 경우, 현재 무게인 w에서 물건 i의 무게인 w_i를 뺀 상태에서 물건을 (i-1)까지 고려했을 때의 최대 가치인 K[i-1,w-w_i]와 물건 i의 가치 v_i의 합이 K[i,w]가 되는 것이다.
- Line 8
  - 이 2가지 경우 중에서 큰 값이 K[i,w]가 된다.

Knapsack 알고리즘

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

배낭에서 물건
i를 담을 공간을
마련

$K[i-1,w]$ 0

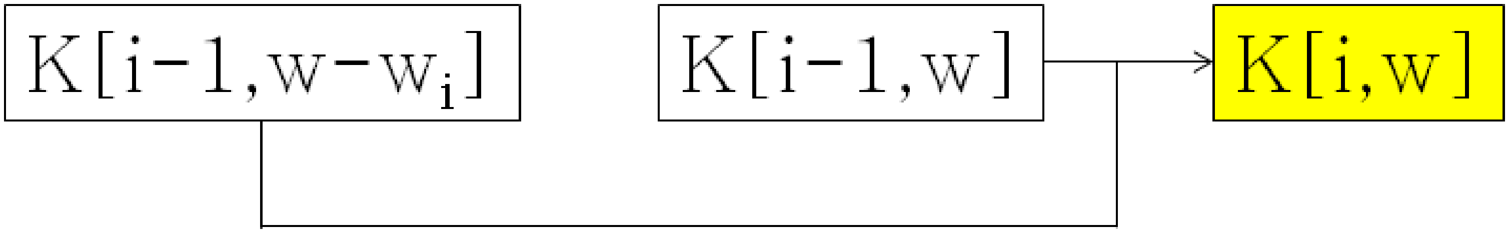
물건 i $\rightarrow$ $K[i,w]$

$K[i-1,w-w_i]$ + 물건 i의 가치 v_i 1

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

Knapsack 알고리즘

- 배낭 문제의 부분 문제간의 함축적 순서는 다음과 같다.
  - 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 수행 과정

- 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

Knapsack 알고리즘 수행 과정

- 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 알고리즘 수행 과정

- Line 3에서는 물건을 하나씩 고려하기 위해서, 물건 번호 i가 1~4까지 변하며, line 4에서는 배낭의 (임시) 용량 w가 1kg씩 증가되어 마지막에 배낭의 용량인 10kg이 된다.

- i=1일 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]=0이다.

Knapsack 알고리즘 수행 과정

○ w=2, 3, 4일 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 늘려 봐도 5kg의 물건 1을 배낭에 담을 수 없다.

Knapsack 알고리즘 수행 과정

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

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

Knapsack 알고리즘 수행 과정

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

Knapsack 알고리즘 수행 과정

○ 다음은 물건 1에 대해서만 배낭의 용량을 1~C까지 늘려가며 알고리즘을

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 알고리즘 수행 과정

- i=2일 때 (즉, 물건 1에 대한 부분 문제들의 해는 i=1일 때 이미 구하였고, 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>w이므로, 즉, 물건 2의 무게가 4kg이므로, 배낭에 담을 수 없다.
    - 따라서 K[2,1]=0, K[2,2]=0, K[2,3]=0이다.

Knapsack 알고리즘 수행 과정

- w=4 (배낭의 용량이 4kg)일 때,
  - 물건 2를 배낭에 담을 수 있다.

$$K[2,4] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$

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

$$= \max\{K[1,4], K[1,0]+40\}$$

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

=

Knapsack 알고리즘 수행 과정

○ w=5 (배낭의 용량이 5kg)일 때,

- 물건 2의 무게가 4kg이므로, 역시 배낭에 담을 수 있다.
- 그러나 이 경우에는 물건 1을 배낭에 담았을 때의 가치와 물건 2를 담았을 때의 가치를 비교하여, 더 큰 가치를 얻는 물건을 배낭에 담는다.

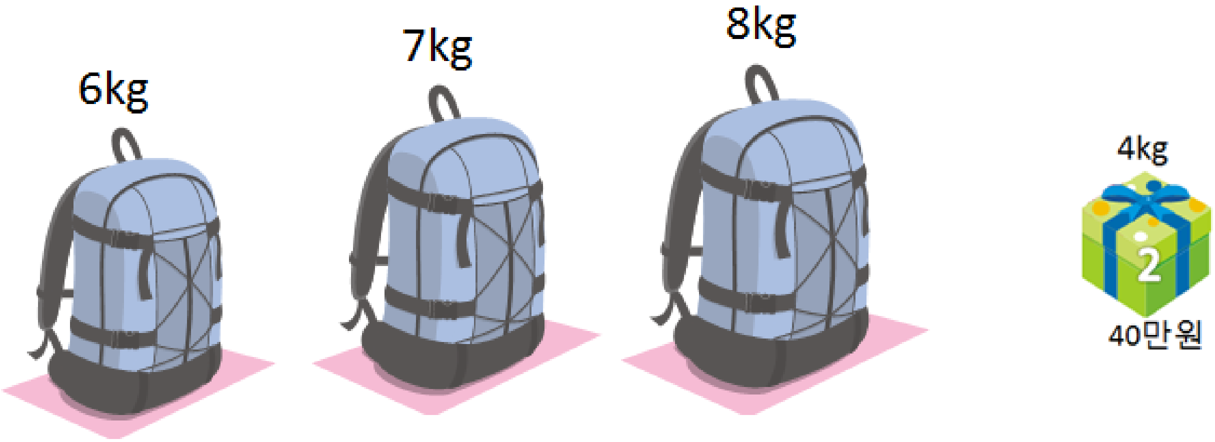
$$K[2,5] = \max\{K[i-1,w], K[i-1,w-w_i]+v_i\}$$
$$= \max\{K[2-1,5], K[2-1,5-4]+40\}$$
$$= \max\{K[1,5], K[1,1]+40\}$$
$$= \max\{10, 0+40\}$$
$$= \max\{10, 40\} = 40\text{이다.}$$

- 즉, 물건 1을 배낭에서 빼낸 후, 물건 2를 담는다.
- 그 때의 가치가 40이다.

Knapsack 알고리즘 수행 과정

○ w=6, 7, 8일 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] = 40이 된다.



Knapsack 알고리즘 수행 과정

○ w=9 (배낭의 용량이 9kg)일 때,

- 물건 2를 배낭에 담아보려고 한다.
- 그런데 w_2<w 이므로, 물건 2를 배낭에 담을 수 있다. 따라서

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

- 즉, 이때에는 배낭에 물건 1, 2 둘 다를 담을 수 있는 것이고, 그때의 가치가 50이 된다는 의미이다.

Knapsack 알고리즘 수행 과정

○ w=10 (배낭의 용량이 10kg)일 때,

- w_2<w 이므로, w=9일 때와 마찬가지로 K[2,10]=50이고,
- 물건 1, 2를 배낭에 둘 다 담을 때의 가치인 50을 얻는다는 의미이다.

○ 다음은 물건 1과 2에 대해서만 배낭의 용량을 1부터 C까지 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	0										

Knapsack 알고리즘 수행 과정

- i=3과 i=4일 때 알고리즘이 수행을 마친 결과

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	$\boxed{90}$

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

시간복잡도

- 하나의 부분 문제에 대한 해를 구할 때의 시간복잡도
  - line 5에서의 무게를 한 번 비교한 후 line 6에서는 1개의 부분문제의 해를 참조하고, line 8에서는 2개의 해를 참조한 계산이므로 O(1) 시간이 걸린다.

- 그런데 부분 문제의 수는 배열 K의 원소 수인 n x C개이다.
  - 여기서 C는 배낭의 용량이다.

- 따라서 Knapsack 알고리즘의 시간복잡도
  - O(1) x n x C = O(nC)

응용

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

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.
  - 예) 160원 동전이 있을 경우, 200원 거스름돈 문제
  - 그리디 알고리즘: 160원*1개, 10원*4개

- 동적 계획 알고리즘은 모든 동전 거스름돈 문제에 대하여 항상 최적해를 찾는다.

동전 거스름돈 문제

- 동적 계획 알고리즘을 고안하기 위해서는 부분 문제를 찾아내야 한다.

- 동전 거스름돈 문제에 주어진 문제 요소들을 생각해보자.
  - 정해진 동전의 종류, d_1, d_2, …, d_k가 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.

- 그런데 배낭 문제의 동적 계획 알고리즘을 살펴보면, 배낭의 용량을 1kg씩 증가시켜 문제를 해결한다.

동전 거스름돈 문제

- 여기서 힌트를 얻어서, 동전 거스름돈 문제도 1원씩 증가시켜 문제를 해결한다.
  - 즉, 거스름돈을 배낭의 용량으로 생각하고, 동전을 물건으로 생각

- 부분 문제들의 해를 아래와 같이 1차원 배열 C에 저장한다.
  - 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 문제

- 부분문제들 사이의 ‘함축적인 순서’, 즉, 한 부분문제의 해를 구하는데 어떤 부분 문제의 해가 필요한지를 살펴보자.

- 구체적으로 C[j]를 구하는데 어떤 부분문제가 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)원 의 해, 즉, C[j-500] = C[j-d_1]에다가 500원짜리 동전 1개를 추가한다.
  - 100원짜리 동전이 거스름돈 j원에 필요하면 (j-100)원 의 해, 즉, C[j-100] = C[j-d_2]에다가 100원짜리 동전 1개를 추가한다.

동전 거스름돈 문제

- 50원짜리 동전이 거스름돈 j원에 필요하면 (j-50)원의 해, 즉, C[j-50] = C[j-d_3]에다가 50원짜리 동전 1개를 추가한다.
- 10원짜리 동전이 거스름돈 j원에 필요하면 (j-10)원의 해, 즉, C[j-10] = C[j-d_4]에다가 10원짜리 동전 1개를 추가한다.
- 1원짜리 동전이 거스름돈 j원에 필요하면 (j-1)원의 해, 즉, C[j-1] = C[j-d_5]에다가 1원짜리 동전 1개를 추가한다.

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

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

- 위의 식에서는 i가 1~k까지 각각 변하면서, 즉, $d_1, d_2, d_3, \cdots, d_k$ 각각에 대하여 해당 동전을 거스름돈에 포함시킬 경우의 동전 수를 고려하여 최소값을 C[j]로 정한다.

DPCoinChange 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]=∞
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 {
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]
```

DPCoinChange 알고리즘

- Line 1
  - 배열 C의 각 원소를 ∞로 초기화 한다.
  - 이는 문제에서 거슬러 받는 최소 동전 수를 구하기 때문
- Line 2
  - C[0]=0으로 초기화한다.
  - 이는 line 5에서 $C[j-d_i]$의 인덱스인 j에서 d_i를 뺀 값이 0이 되는 경우, 즉, C[0]이 되는 경우를 위해서이다.
- Line 3~6
  - for-루프에서는 (임시) 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
  - for-루프에서는 가장 큰 액면의 동전부터 1원짜리 동전까지 차례로 동전을 고려해보고, 그 중에서 가장 적은 동전 수를 C[j]로 결정한다.
  - 단, 거스름돈 액수인 j원보다 크지 않은 동전에 대해서만 고려한다.

DPCoinChange 알고리즘 수행 과정

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

DPCoinChange 알고리즘 수행 과정

- Line 1~2
  - 배열 C를 아래와 같이 초기화 시킨다.

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

- 거스름돈 j원은 1원~4원까지
  - 1원짜리 동전 (d_4=1)밖에 고려할 동전이 없으므로, 각 j에서 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≤j)가 '참'이고, (C[j-1]+1<∞)도 '참'이 되어 각각 아래와 같이 C[j]가 결정된다.

DPCoinChange 알고리즘 수행 과정

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

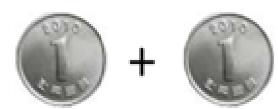
j	0	1
	0	∞

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

DPCoinChange 알고리즘 수행 과정

- j=5이면 임시 거스름돈이 5원일 때
  - i=3 (5원짜리 동전)에 대해서, line 5의 if-조건인 (5≤5)가 ‘참’이고,
  - (C[5-5]+1 < C[5]) = (C[0]+1 <∞) = (0+1 <∞)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 = 1이 된다. 즉, C[5]=1이다.

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

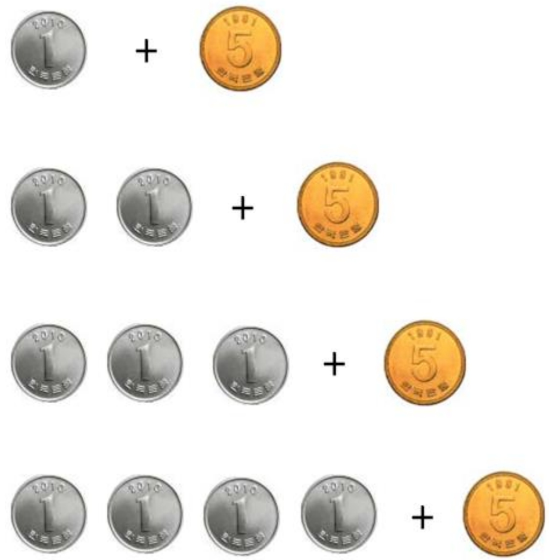
  - i=4 (1원짜리 동전)일 때는 line 5의 if-조건인 (d_4 ≤ 5)는 ‘참’이나 (C[j-d_i]+1 < C[j]) = (C[5-1] +1 < C[5]) = (C[4] +1 < C[5]) = (4+1 < 1) = (5 < 1)가 ‘거짓’이 되어 C[5]는 변하지 않고 그대로 1을 유지한다.
  - 즉, 1원짜리 동전으로 거스름돈을 주려 하면 오히려 동전 수가 늘어나기 때문이다.

DPCoinChange 알고리즘 수행 과정

- j=6, 7, 8, 9이고, i=3 (5원짜리 동전)일 때, 각각 아래와 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

- 단, i=4 (1원짜리 동전)일 때에는 line 5의 if-조건의 ($C[j-d_i]+1 < C[j]$)= (C[j-1]+1 < C[j])이 각각의 j에 대해서 (1+1) < 2, (2+1) < 3, (3+1) < 4, (4+1) < 5로서 ‘거짓’이 되어 C[j]는 변경되지 않는다. 사실은 i=3일 때와 동일하므로 각각 갱신 안 된다.

DPCoinChange 알고리즘 수행 과정

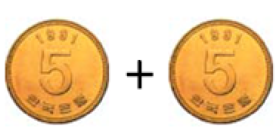


j	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	1	∞	∞	∞	∞
	0	1	2	3	4	1	**2**	∞	∞	∞
C	0	1	2	3	4	1	2	**3**	∞	∞
	0	1	2	3	4	1	2	3	**4**	∞
	0	1	2	3	4	1	2	3	4	**5**

DPCoinChange 알고리즘 수행 과정

- j=10, 거스름돈이 10원이면
  - i=2 (10원짜리 동전)일 때, line 5의 if-조건인 $(d_i \le j)=(10 \le 10)$은 '참'이고, $(C[j-d_i]+1<C[j]) = (C[10-10]+1<C[10]) = (C[0]+1< C[10]) = (0+1<\infty)$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 = 1이다. 즉, C[10]=1이다.

  - i=3 (5원짜리 동전)일 때, line 5의 if-조건인 $(d_i \le j) = (5<10)$는 '참'이나, $(C[10-5]+1<C[10]) = (C[5]+1<C[10]) = (1+1<1)$이 '거짓'이 되어서 C[10]은 변하지 않는다. 즉, 5원짜리 2개보다는 10원짜리 1개 낫다.



DPCoinChange 알고리즘 수행 과정

- i=4 (1원짜리 동전)일 때는 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) = (6<1)이 '거짓'이므로 C[10]이 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 알고리즘 수행 과정

- j=20일 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-10]+1 = C[10]+1 = 1+1 = 2이므로 현재 C[20]의 값인 5보다 작다. 따라서 if-조건이 ‘참’이 되어 C[20]=2가 된다.

  - i=3 (5원짜리 동전)일 때에는 line 5의 if-조건의 (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’이 되어 C[20]이 변경되지 않는다.

DPCoinChange 알고리즘 수행 과정

- i=4 (1원짜리 동전)일 때에도 line 5의 if-조건이 (C[20-1]+1<2) = (5<2)이 ‘거짓’이므로 C[20]이 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

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

DPCoinChange 알고리즘 수행 과정

- 따라서 거스름돈 20원에 대한 최종해는 C[20]=2개의 동전이다.

- 그리디 알고리즘은 20원에 대해 16원짜리 동전을 먼저 '욕심내어' 취하고, 4원이 남게 되어, 1원짜리 4개를 취하여, 모두 5개의 동전이 해라고 답한다.

그리디 알고리즘의 해

동적 계획 알고리즘의 해

시간복잡도

- DPCoinChange 알고리즘의 시간복잡도
  - O(nk)
  - 이는 거스름돈 j가 1원~n원까지 변하며, 각각의 j에 대해서 최악의 경우 모든 동전 (d_1, d_2, …, d_k)을 (즉, k개를) 1번씩 고려하기 때문이다.

요약

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

- 동적 계획 알고리즘에는 부분문제들 사이에 의존적 관계가 존재한다.

요약

- 모든 쌍 최단 경로 (All Pairs Shortest Paths) 문제를 위한 Floyd-Warshall 알고리즘은 $O(n^3)$ 시간에 해를 찾는다.

- 핵심 아이디어는 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 계산하는 것이다.

요약

- 연속 행렬 곱셈 (Chained Matrix Multiplications) 문제를 위한 $O(n^3)$ 시간 동적 계획 알고리즘의 아이디어는 주어진 연속된 행렬들의 순서를 지켜서 이웃하는 행렬들끼리 곱하는 모든 부분 문제들을 해결하는 것이다.

- 배낭 (Knapsack) 문제를 위한 동적 계획 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 변화시켜가며 해를 찾는다. 시간 복잡도는 O(nC)이다.

- 동전 거스름돈 (Coin Change)문제는 1원씩 증가시켜 문제를 해결한다. 배낭 문제와 유사한 문제로서 거스름돈을 배낭의 용량으로 생각하고, 동전을 물건이라고 생각하면 된다. 시간복잡도는 O(nk)이다. 단, n은 거스름돈 액수이고, k는 동전 종류의 수이다.

요약

- 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.