ALOHA
#6주차

Knapsack & Interval DP

#CH.0

Knapsack 문제란?

Knapsack 문제란?

배낭에 담을 수 있는 무게의 최댓값이 정해져 있고
일정 **가치**와 **무게**가 있는 짐들을 배낭에 넣을 때,
가치의 합이 최대가 되도록 짐을 고르는 방법을 찾는 문제

Knapsack 문제의 종류

Fractional Knapsack Problem

보석을 쪼개서 가방에 담을 수 있는 경우
보석들 중 무게 대비 가격이 가장 높은 것부터 담으면 된다.

0-1 Knapsack Problem

여러 종류의 보석들이 각각 한 개씩만 있는 경우
가장 대표적인 Knapsack 문제

Unbounded Knapsack Problem

여러 종류의 보석들이 무한히 존재하며 원하는 만큼 담을 수 있는 경우

#CH.1

0-1 Knapsack Problem

0-1 Knapsack Problem

DP 배열의 정의

DP[k][w] = “크기 w의 배낭 안에 1~k번 보석까지 확인하여 담았을 때 최대 가치”

배낭의 최대 무게를 M이라고 했을 때, k번째 보석까지 확인하여 DP[k][M] 배열까지 채웠다면,
k+1번째 보석을 배낭에 넣어 DP[k][w]를 채울 때 두 가지 상황을 가정할 수 있다.

- k+1번째 보석을 담는 경우
- k+1번째 보석을 담지 않는 경우

0-1 Knapsack Problem

k+1번 보석을 담는 경우

가방에 k+1번 보석이 들어가야 하므로, 무게를 W' 라고 할 때, DP[k][w-W']에서
k+1번째 보석의 가치(c)를 더해준 값이 DP[k+1][w]의 값이 된다.

K+1번 보석을 담지 않는 경우

가방에 들어있는 보석은 변하지 않으므로, DP[k+1][w]와 DP[k][w]의 값은 같다.

0-1 Knapsack Problem

k+1번 보석을 담는 경우

가방에 k+1번 보석이 들어가야 하므로, 무게를 W' 라고 할 때, DP[k][w-W']에서
k+1번째 보석의 가치(c)를 더해준 값이 DP[k+1][w]의 값이 된다.

이 두 가지 경우에서 가능한 DP[k+1][w]의 값들 중, 최댓값을 대입해준다!

K+1번 보석을 담지 않는 경우

가방에 [illegible]

DP[k+1][w] = max(DP[k][w], DP[k][w-W'] +c)

0-1 Knapsack Problem

```
int N, W;                           // N : 보석의 개수
                                    // W : 배낭의 크기
int _w[N_Max + 1], c[N_Max + 1];//_w[k] : k번째 보석의 무게
                                    //c[k] : k번째 보석의 무게
int DP[N_Max + 1][W_Max + 1];
for (int k = 0; k < N; k++) {
    for (int w = 1; w <= W; w++) {
        if (w >= _w[k])             // w - _w[k]가 음수가 되지 않도록 체크!
            DP[k+1][w] = max(DP[k][w], DP[k][w - _w[k]] + c[k+1]);
        else
            DP[k+1][w] = DP[k][w];
    }
}
```

풀어볼까유

#1535 안녕

다음 장에 풀이

0-1 Knapsack Problem

전형적인 0-1 Knapsack 문제

배낭의 최대 용량을 99 (체력이 0이 되면 죽으므로)

인사할 때 잃는 체력을 보석의 무게,

인사할 때 얻는 기쁨을 보석의 가치로 생각한다.

0-1 Knapsack Problem

```
int N, W;                          // N : 보석의 개수
                                   // W : 배낭의 크기
int _w[N_Max + 1], c[N_Max + 1];//_w[k] : k번째 보석의 무게
                                   //c[k] : k번쨰 보석의 무게
int DP[N_Max + 1][W_Max + 1];
for (int k = 0; k < N; k++) {
    for (int w = 1; w <= W; w++) {
        if (w >= _w[k])            // w - _w[k]가 음수가 되지 않도록 체크!
            DP[k+1][w] = max(DP[k][w], DP[k][w - _w[k]] + c[k+1]);
        else
            DP[k+1][w] = DP[k][w];
    }
}
```

위 코드에서 N과 N_Max를 사람의 수, W와 W_Max를 99라고 하고
_w 배열과 c배열에 각각 인사했을 때 잃는 체력과 얻는 기쁨들을 입력받으면,

결과적으로 DP[N_Max][W_Max]칸의 값이 우리가 원하는 답이 된다.

#CH.2

Unbounded Knapsack Problem

Unbounded Knapsack Problem

DP 배열의 정의

DP[k][w] = “크기 w의 배낭 안에 1~k번 보석까지 확인하여 담았을 때 최대 가치”

배낭의 최대 무게를 M이라고 했을 때, k번째 보석까지 확인하여 DP[k][M] 배열까지 채웠다면,
k+1번째 보석을 배낭에 넣어 DP[k][w]를 채울 때 두 가지 상황을 가정할 수 있다.

- k+1번째 보석을 1개 이상 담는 경우
- k+1번째 보석을 담지 않는 경우

Unbounded Knapsack Problem

k+1번 보석을 1개 이상 담는 경우

가방에 k+1번 보석이 들어가야 하므로, 무게를 W' 라고 할 때, DP[k+1][w-W']에서
k+1번째 보석의 가치(c)를 더해준 값이 DP[k+1][w]의 값이 된다.

K+1번 보석을 담지 않는 경우

가방에 들어있는 보석은 변하지 않으므로, DP[k+1][w]와 DP[k][w]의 값은 같다.

※ DP[k+1][w-W'] = w-W' 크기의 배낭에 k+1번째 보석까지 확인하여 담았을 때, 담을 수 있는 최대 가치를 저장한 칸

Unbounded Knapsack Problem

k+1번 보석을 1개 이상 담는 경우

가방에 k+1번 보석이 들어가야 하므로, 무게를 W' 라고 할 때, DP[k+1][w-W']에서
k+1번째 보석의 가치(c)를 더해준 값이 DP[k+1][w]의 값이 된다.

무엇이 달라졌을까?

K+1번 보석을 담지 않는 경우

가방에 들어있는 보석은 변하지 않으므로, DP[k+1][w]와 DP[k][w]의 값은 같다.

※ DP[k+1][w-W'] = w-W' 크기의 배낭에 k+1번째 보석까지 확인하여 담았을 때, 담을 수 있는 최대 가치를 저장한 칸

Unbounded Knapsack Problem

k+1번 보석을 1개 이상 담는 경우

0-1 Knapsack의 경우, k+1번째 보석을 추가로 담을 때, k+1번째 보석이 가방에 없었다는 가정이 있다. 즉, DP[k][w-W'] 배열을 참조해야 한다.

Unbounded Knapsack의 경우, k+1번째 보석을 추가로 담을 때, 가방 안에 k+1번째 보석이 이미 담겨 있을 수 도 있다. 즉 DP[k+1][w-W'] 배열을 참조해야 한다.

Unbounded Knapsack Problem

k+1번 보석을 1개 이상 담는 경우

0-1 Knapsack의 경우, k+1번째 보석을 추가로 담을 때, k+1번째 보석이 가방에 없었다는 가정이 있다. 즉, DP[k][w-W'] 배열을 참조해야 한다.

Unbounded Knapsack의 경우, 보석을 추가로 담을 때, 기존 안에 k+1번째 보석이 이미 담겨 있을 수 도 있다. 즉 DP[k+1][w-W'] 배열을 참조해야 한다.

즉, DP[k][w-W']을 참조하느냐, DP[k+1][w-W']을 참조하느냐 외에는 구현하는 데 차이점이 없다.

Unbounded Knapsack Problem

0-1 Knapsack의 경우 DP배열의 크기는 N*W이고, 배열을 모두 채워야 하므로, 시간복잡도 또한 O(NW)이다.

Unbounded Knapsack의 경우도 마찬가지로, 공간복잡도와 시간복잡도 모두 O(NW)이다.

Unbounded Knapsack Problem

공간 복잡도 줄이기

Unbounded Knapsack의 경우, 0-1 Knapsack과 달리, 1차원 배열로 구현할 수 있다.

DP[k+1][w]를 구하기 위해 필요한 곳은 DP[k][w]와 DP[k][w-W']이다.
이를 DP[W_max]를 이용해 똑같이 구현할 수 있다.

Unbounded Knapsack Problem

EX) 총 배낭의 용량: 8
1번째 보석의 무게: 2, 1번째 보석의 가격: 4
2번째 보석의 무게: 3, 2번째 보석의 가격: 7

DP	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0

Unbounded Knapsack Problem

1번째 보석까지 확인하여 테이블을 채우면 다음과 같다.

1번째 보석의 무게: 2, 1번째 보석의 가격: 4

DP	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0

Unbounded Knapsack Problem

1번째 보석까지 확인하여 테이블을 채우면 다음과 같다.

1번째 보석의 무게: 2, 1번째 보석의 가격: 4

DP	0	1	2	3	4	5	6	7	8
	0	0	4	0	0	0	0	0	0

Unbounded Knapsack Problem

1번째 보석까지 확인하여 테이블을 채우면 다음과 같다.

1번째 보석의 무게: 2, 1번째 보석의 가격: 4

DP	0	1	2	3	4	5	6	7	8
	0	0	4	4	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7
2번째 보석의 무게가 3이므로, DP[3]부터 갱신해 나간다.

DP[3]을 갱신하는 과정에서 이미 DP[3]에 들어간 값 '4'는 1번째 보석까지 확인했을 때의 최대 가치이다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	4	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

DP[3]에 원래 있던 값인 4와 DP[3-3]+7을 비교해준다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	4	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

DP[3]에 원래 있던 값인 4와 DP[3-3]+7을 비교해준다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	8	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	12	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

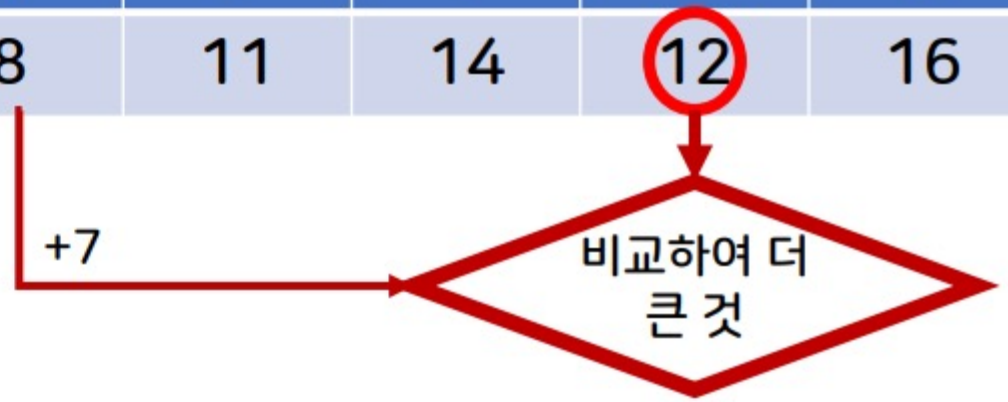
DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	14	12	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	14	12	16



Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	14	15	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	14	15	16

Unbounded Knapsack Problem

2번째 보석의 무게: 3, 2번째 보석의 가격: 7

이를 반복하여 DP[8]까지 갱신한다.

DP	0	1	2	3	4	5	6	7	8
	0	0	4	7	8	11	14	15	18

풀어볼까유

#4781 사탕가게

대표적인 문제

풀어볼까유

#9084 동전

대표적인 문제

#CH.3

Interval DP

Interval DP란?

2차원 DP의 일종

DP[i][j]가 i번째부터 j번째까지의 정보를 담고 있을 때, 이러한 종류의 DP를 Interval DP라고 한다.

풀어볼까유

#11066 파일 합치기

대표적인 문제

Interval DP - boj.kr/11066

DP table의 정의

처리해야 하는 배열은 문자열 하나, DP table은 2차원으로 정의

DP[i][j]는 i번째 파일부터 j번째 파일을 합칠 때 필요한 최소 비용을 저장

문제의 예제 입력에서 파일 4개 (40, 30, 30, 50)을 합치는 경우를 살펴보자

Interval DP - boj.kr/11066

DP table의 정의에 따라 우리가 원하는 답은 DP[1][4]에 저장되어 있다.

마지막 1개의 파일을 만들 때, 아래 3가지 경우가 있다.

{40}/{30,30,50}
{40,30}/{30,50}
{40,30,30}/{50}

Interval DP - boj.kr/11066

{40}/{30,30,50}의 경우를 살펴보자.

{40}이라는 파일의 경우, 다른 파일과 합친 적이 없으므로, {40}이라는 파일을 만드는데 드는 비용은 00이다.
즉, DP[1][1] 은 00이다.

{30,30,30}의 파일을 만드는 최소 비용도 앞 슬라이드와 같이 2가지로 생각할 수 있다.
{30}/{30,50}
각각 0, 80의 비용이 들고, {30}과 {30,50}을 합치는데 110의 비용이 드므로 총 190
{30,30}/{50}
각각 60, 0의 비용이 들고, {30,30}과 {50}을 합치는데 110의 비용이 드므로 총 170

위의 두 경우를 비교했을 때, {30,30,50}의 파일을 만드는 데 최소 170의 비용이 필요함을
알 수 있다.

Interval DP - boj.kr/11066

{40}/{30,30,50}의 경우를 살펴보자.

{40}이라는 파일의 경우, 다른 파일과 합친 적이 없으므로, {40}이라는 파일을 만드는데 드는 비용은 0이다.

즉, DP[1][1] = 0 이다.

{30,30,30}의 파일을 만드는 데 드는 최소 비용을 구하는 방법은 2가지가 있다.

{30}/{30,50}

각각 0, 150의 비용이 들고, {30}과 {50}을 합치는데 90의 비용이 드므로 총 240

{30,30}/{50}

각각 60, 0의 비용이 들고, {30,30}과 {50}을 합치는데 110의 비용이 드므로 총 170

위의 두 경우를 비교했을 때, {30,30,50}의 파일을 만드는 데 최소 170의 비용이 필요함을 알 수 있다.

즉, {40}과 {30,30,50} 두 파일을 합쳐서 {40,30,30,50} 파일을 만드는 데는 0 + 170 + 150 = 320의 비용이 드는 것을 알 수 있다.

Interval DP - boj.kr/11066

마찬가지 방식으로 다른 경우의 비용을 구해보면
{40,30}/{30,50}은 300, {40,30,30}/{50}의 경우 310의 비용이 드는 것을 알 수 있다.

이 3가지 경우 중 최소 비용이 드는 것은 {40,30}/{30,50}이므로, DP[1][4] = 3000이다.

즉, DP[i][j] = min(i<=k<j){DP[i][k] + DP[k+1][j]} + (sum[j] - sum[i-1])

어떻게 쪼개진 파일을 합쳐야 최소 비용이 되는지 결정

쪼개진 파일들을 합칠 때 드는 비용(일정)

sum[i]는 0~i번째 배열까지의 합을 의미함

다음 시간에 만나요~