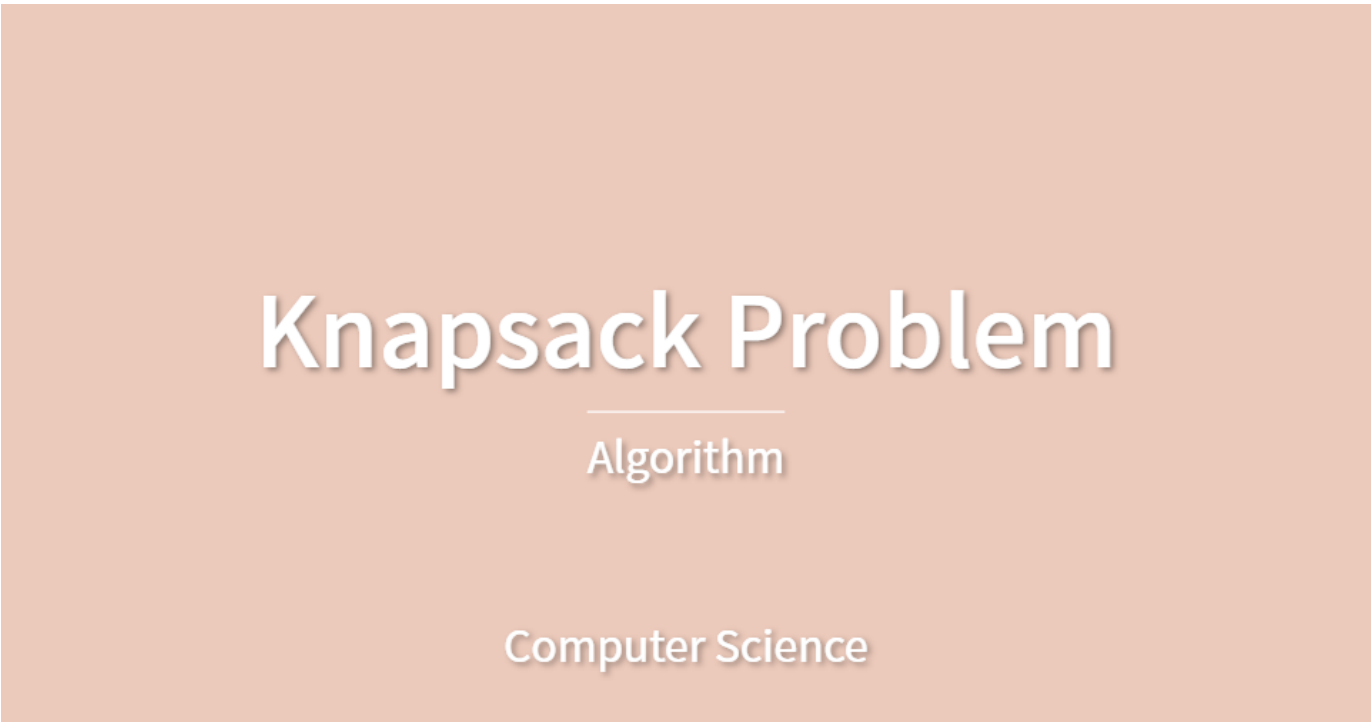
[DP] 배낭 문제 (Knapsack Problem)

jxlhe46 · 2021년 11월 13일

1

알고리즘



알고리즘 이해

> 배낭 문제는 **n개의 물건**과 **각 물건 i의 무게 wi와 가치 vi**가 주어지고, **배낭의 용량이 C**일 때, **배낭에 담을 수 있는 물건의 최대 가치를 찾는 문제**이다. 단, 배낭에 담은 물건의 무게의 합이 C를 초과하지 말아야 하고, 각 물건은 1개씩만 있다고 가정한다.

가장 기본적으로 모든 경우의 수를 다 따져보면 어떻게 될까? (브루트포스)
n개의 물건 각각에 대해서 배낭에 담거나 담지 않거나 이 두 가지 경우가 있기 때문에 시간복잡도는 O(2^n)이고, 너무 오래 걸린다.

그렇다면 n개의 물건에 대해서 '단위 무게당 가치가 가장 높은 물건부터' 욕심내어 배낭에 담는 것은 어떨까? (그리디)

물건을 부분적으로 담는 것이 허용되는 **부분 배낭 문제 (Fractional Knapsack Problem)** 에서는 이와 같은 그리디 알고리즘으로 최적해를 구할 수 있다. 예를 들어, 물건이 금, 은, 백금 등과 같은 분말이라고 가정하면, 원하는 무게만큼만 배낭에 담을 수 있다.

하지만, 물건을 통째로 담는 것만 허용되는 **0-1 배낭 문제**에서는 그리디 알고리즘으로 최적해를 구할 수 없다. (물건을 배낭에 담지 않으면 '0', 담으면 '1'로 본다.)

예를 들어, 배낭의 용량이 30kg이고 각 물건의 무게와 가치가 다음과 같다고 하자.

> 물건 1: 5kg, $5 → kg당 $1
> 물건 2: 10kg, $6 → kg당 $0.6
> 물건 3: 20kg, $14 → kg당 $0.7

단위 무게당 가치가 가장 높은 물건부터 배낭에 담으면 물건 1, 3이 담기기 때문에 배낭에는 5kg의 공간이 남고 얻은 가치는 $19이다. 하지만 실제 최적해는 물건 2, 3을 담은 경우이다. 이때는 배낭에 남은 공간이 없고, 얻은 가치는 $20이다.

부분 배낭 문제에서는 물건 1, 3을 넣고 남은 공간에 물건 2를 5kg만 부분적으로 담을 수 있기 때문에, $(5 + 14 + 5 * 0.6) = $22로 최적해를 구할 수 있다. 하지만 물건을 통째로만 넣을 수 있는 0-1 배낭 문제에서는 이 방법으로 최적해를 구할 수 없다. 이때 사용할 수 있는 방법이 다이나믹 프로그래밍이다.

다이나믹 프로그래밍은 문제가 다음 조건을 만족시킬 때 사용할 수 있다.

1. 최적 부분 구조 (Optimal substructure): 큰 문제를 작은 문제로 나눌 수 있으며, 작은 문제의 답을 모아서 큰 문제를 해결할 수 있다.
2. 중복되는 부분 문제 (Overlapping subproblem): 동일한 작은 문제를 반복적으로 해결한다.

배낭 문제의 부분 문제를 정의하기 위해 물건은 하나씩 차례로 고려하면 되지만, 물건의 무게가 각각 다를 수 있기 때문에 무게에 대해서는 **배낭의 용량이 0(kg)부터 1(kg)씩 증가하여 입력으로 주어진 용량인 C(kg)이 될 때까지 변화시켜가면서,**

물건을 배낭에 담는 것이 가치가 더 커지는지를 결정해야 한다.
그래서 원래 배낭의 용량은 C(kg)이지만, 배낭 용량이 0(kg)부터 1(kg)씩 증가할 경우의 용량을 '임시' 배낭 용량이라고 부르면, **배낭 문제의 부분 문제**를 다음과 같이 정의할 수 있다. 여기서 **K는 배낭에 담을 수 있는 물건의 최대 가치를 저장하는 2차원 테이블**이다.

> K[i, w] = 물건 1~i까지만 고려하고, (임시) 배낭 용량이 w일 때의 최대 가치
> (단, i = 1, 2, ..., n, w = 1, 2, ..., C)

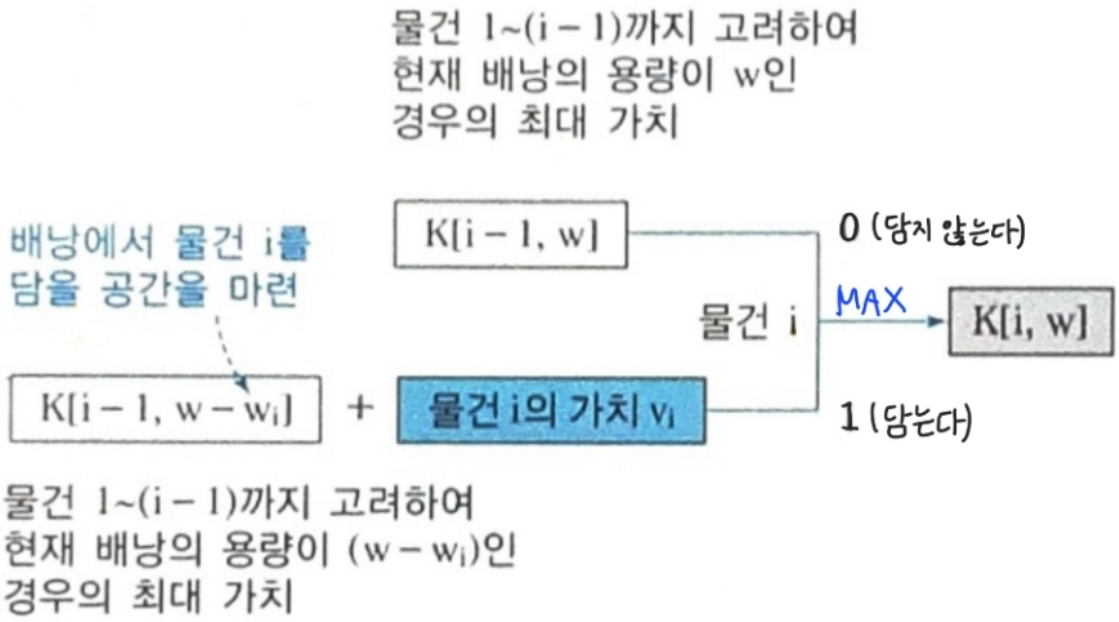
그러므로 **문제의 최적해는 K[n, C]**이다.

의사코드

```
입력: 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 wi, 가치 vi (단, i = 1, 2, ..., n)
출력: K[n, C]

// 배낭의 용량이 0일 때, 어떤 물건도 담을 수 없으므로 가치 = 0
1. for i = 0 to n K[i, 0] = 0

// 물건이 0이라는 건 어떤 물건도 배낭에 담으려고 고려하지 않았을 때를 의미함.
2. for w = 0 to C K[0, w] = 0

3. for i = 1 to n { // 물건 1~n
4.  for w = 1 to C { // 배낭의 임시 용량 1~C
5.      if(wi > w){ // 물건 i의 무게가 배낭의 임시 용량을 초과하면
6.          K[i, w] = K[i − 1, w] // 물건 i−1 까지만 담는다.
7.      else // wi <= w
8.          K[i, w] = max{K[i − 1, w], K[i − 1, w − wi] + vi}
        }
    }
   }
9. return K[n, C] // 최적해 리턴
```

line7은 현재 고려하는 물건 i의 무게 wi가 현재 배낭의 임시 용량 w 이하여서, 물건 i를 배낭에 담을 수 있다. 그러나, 현재 상태에서 물건 i를 추가로 배낭에 담으면, 배낭의 무게가 (w + wi)로 늘어난다. 따라서, line8에서는 다음 2가지 경우 중에 더 큰 값으로 K[i, w]를 설정해줘야 한다.

> 1) 물건 i를 배낭에 담지 않는 경우, K[i, w] = K[i - 1, w]
> 2) 물건 i를 배낭에 담는 경우, 현재 무게인 w에서 물건 i의 무게인 wi를 뺀 상태에서 물건을 (i-1)까지 고려했을 때의 최대 가치인 K[i-1, w-wi]와 물건 i의 가치 vi의 합이 K[i, w]가 된다.



이처럼 배낭 문제는 K라는 2차원 테이블을 채워가면서, 필요한 경우 **"앞에서 계산했던 다른 칸의 값을 이용해" 다음 칸의 값을 계산**하므로 DP 문제로 볼 수 있다. 여기서 다음과 같은 DP 문제의 조건을 다시 상기시켜보자.

1. 최적 부분 구조 (Optimal substructure): 큰 문제를 작은 문제로 나눌 수 있으며, **작은 문제의 답을 모아서 큰 문제를 해결**할 수 있다.
2. 중복되는 부분 문제 (Overlapping subproblem): **동일한 작은 문제를 반복적으로 해결**한다.

그리고 K[i, w]는 K[i - 1, w - wi]와 K[i - 1, w]가 미리 계산되어 있어야만 구할 수 있으므로 **부분문제 사이에 의존적인 관계가 존재한다**는 것도 확인할 수 있다.

예제

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

배낭의 용량 C = 10kg일 때, 배낭에 담을 수 있는 물건의 최대 가치는 얼마인지 구해보자.

부분문제의 정의는 다음과 같다는 걸 기억하자!

K[i, w] = 물건 1~i까지만 고려하고, (임시) 배낭 용량이 w일 때의 최대 가치
(단, i = 1, 2, ..., n, w = 1, 2, ..., C)

먼저, i = 0, w = 0인 경우 테이블의 0번 행과 0번 열의 각 원소를 0으로 초기화 시킨다.

배낭 용량 w =			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건 i	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										

i = 1일 때, 물건 1만 고려한다.

물건 1의 무게가 5kg이기 때문에 w = 1~4일 때는 물건 1을 담을 수 없다. w = 5일 때 비로소 물건 1을 담을 수 있다.
점화식으로 이해하면 다음과 같다.

K[1, 5] = max{K[i - 1, w], K[i - 1, w - wi] + vi}
= max{K[0, 5], K[0, 5-5] + 10}
= max(0, 10) = 10

물건 1만 고려하기 때문에 w = 6~10일 때도 배낭에 담을 수 있는 최대 가치는 10이다.

배낭 용량 w =			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건 i	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										
6	30	3	0										
3	50	4	0										

i = 2일 때는, 앞서 구했던 물건 1에 대한 부분문제들의 해를 이용하여, 물건 2에 대한 부분문제들의 해를 구한다.

물건 2의 무게가 4kg이기 때문에 w = 4부터 물건을 담을 수 있다.

K[2, 4] = max{K[i - 1, w], K[i - 1, w - wi] + vi}
= max{K[1, 4], K[1, 4-4] + 40}
= max(0, 40) = 40

w = 5일 때는 물건 1을 배낭에 담았을 때의 가치와 물건 2를 배낭에 담았을 때의 가치를 비교하여, 더 큰 가치를 얻는 물건을 배낭에 담는다. 이 경우 물건 1을 빼낸 후 물건 2를 담는다.

K[2, 5] = max{K[i - 1, w], K[i - 1, w - wi] + vi}
= max{K[1, 5], K[1, 5-4] + 40}
= max(10, 40) = 40

w = 6~8도 마찬가지로 물건 2를 담는 것이 더 큰 가치를 얻는다.

w = 9일 때는, 물건 1과 2를 모두 담을 수 있다. w = 10일 때도 마찬가지이다.

K[2, 9] = max{K[i - 1, w], K[i - 1, w - wi] + vi}
= max{K[1, 9], K[1, 9-4] + 40}
= max(10, 10 + 40) = 50

배낭 용량 w =			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건 i	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										
3	50	4	0										

물건 3, 4에 대해서도 같은 계산을 반복해주면 결과적으로 다음과 같은 테이블을 얻을 수 있다.

배낭 용량 w =			0	1	2	3	4	5	6	7	8	9	10
무게	가치	물건 i	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**

따라서 **최적해 K[4, 10]은 물건 2와 4의 가치의 합인 90**이다.

시간복잡도 분석

```
입력: 배낭의 용량 C, n개의 물건과 각 물건 i의 무게 wi, 가치 vi (단, i = 1, 2, ..., n)
출력: K[n, C]

// 배낭의 용량이 0일 때, 어떤 물건도 담을 수 없으므로 가치 = 0
1. for i = 0 to n K[i, 0] = 0

// 물건이 0이라는 건 어떤 물건도 배낭에 담으려고 고려하지 않았을 때를 의미함.
2. for w = 0 to C K[0, w] = 0

3. for i = 1 to n { // 물건 1~n
4.  for w = 1 to C { // 배낭의 임시 용량 1~C
5.      if(wi > w){ // 물건 i의 무게가 배낭의 임시 용량을 초과하면
6.          K[i, w] = K[i - 1, w] // 물건 i-1 까지만 담는다.
7.      else // wi <= w
8.          K[i, w] = max{K[i - 1, w], K[i - 1, w - wi] + vi}
        }
    }
   }
9. return K[n, C] // 최적해 리턴
```

line5에서 무게를 비교한 뒤에, line6에서는 1개의 부분문제의 해를, line8에서는 2개의 부분문제의 해를 참조하여 K[i, w]를 계산하므로, O(1)의 시간이 걸린다. 그런데 **부분문제의 개수**는 2차원 테이블 K의 크기인 n * C이므로, **배낭 문제의 시간복잡도는 O(nC)** 이다.

여기서 배낭의 용량 C가 물건의 개수 n에 비해 너무 커지면, 알고리즘의 수행 시간이 너무 길어져 현실적으로 해를 찾을 수 없다. 따라서 **배낭 문제는 제한적인 입력 크기에 대해서만 효용성을 갖는다.**

백준 12865번. 평범한 배낭

문제

https://www.acmicpc.net/problem/12865

이 문제는 아주 평범한 배낭에 관한 문제이다.

한 달 후면 국가의 부름을 받게 되는 준서는 여행을 가려고 한다. 세상과의 단절을 슬퍼하며 최대한 즐기기 위한 여행이기 때문에, **가지고 다닐 배낭 또한 최대한 가치 있게** 싸려고 한다.

준서가 여행에 필요하다고 생각하는 **N개의 물건**이 있다. **각 물건은 무게 W와 가치 V를 가지는데, 해당 물건을 배낭에 넣어서 가면 준서가 V만큼 즐길 수 있다.** 아직 행군을 해본 적이 없는 준서는 **최대 K만큼의 무게만을 넣을 수 있는 배낭만 들고 다닐 수 있다.**

준서가 최대한 즐거운 여행을 하기 위해 **배낭에 넣을 수 있는 물건들의 가치의 최댓값**을 알려주자.

입력

첫 줄에 **물품의 수 N(1 ≤ N ≤ 100)**과 준서가 **버틸 수 있는 무게 K(1 ≤ K ≤ 10만)**가 주어진다.

두 번째 줄부터 N개의 줄에 거쳐 **각 물건의 무게 W(1 ≤ W ≤ 10만)와 해당 물건의 가치 V(0 ≤ V ≤ 1,000)**가 주어진다.

입력으로 주어지는 모든 수는 정수이다.

출력

한 줄에 **배낭에 넣을 수 있는 물건들의 가치합의 최댓값**을 출력한다.

예제

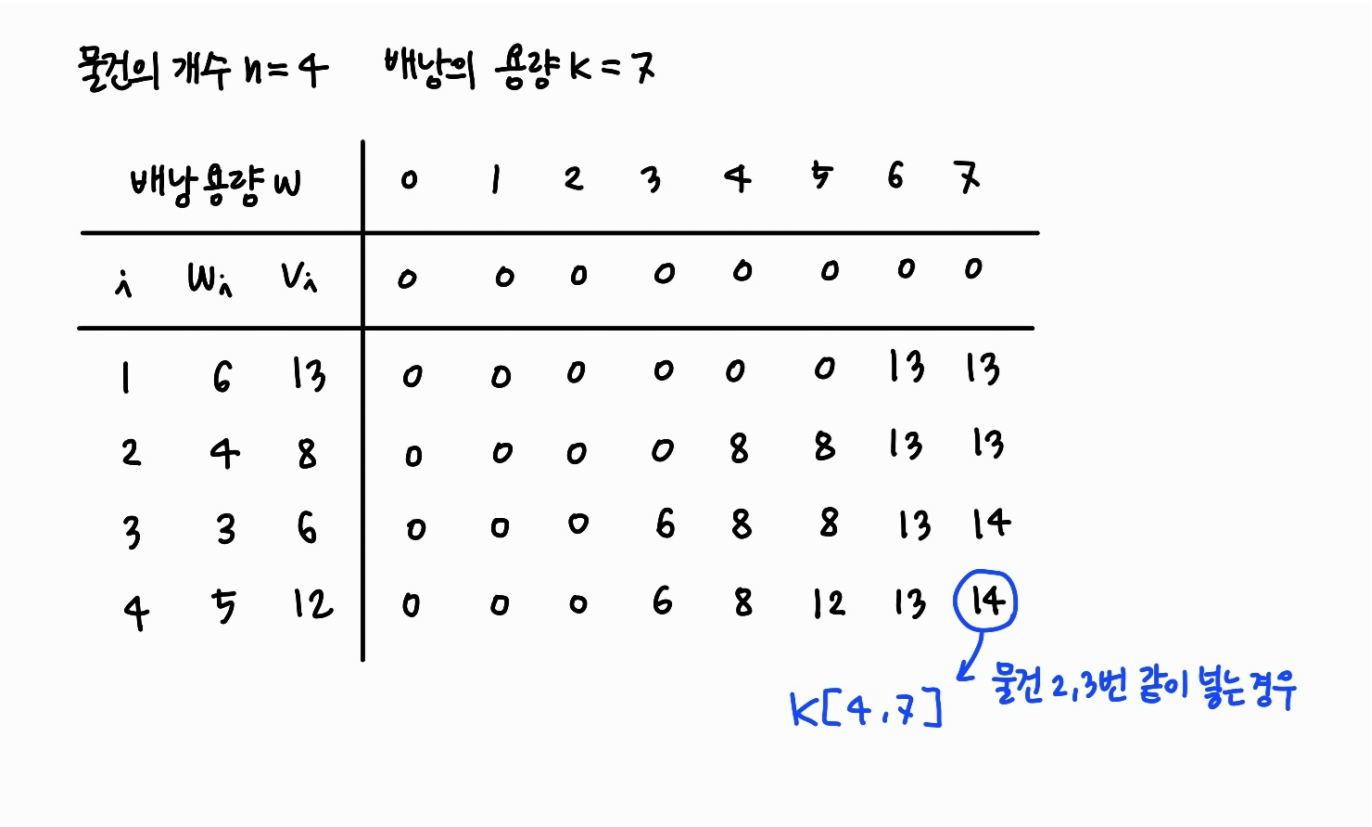
예제 입력 1 복사

```
4 7
6 13
4 8
3 6
5 12
```

예제 출력 1 복사

```
14
```

풀이



```cpp
#include <iostream>
using namespace std;

int DP[101][100001];
int w[101];
int v[101];

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    // n: 물건의 수, k: 배낭의 용량
    int n, k;
    cin >> n >> k;

    // 각 물건의 무게와 가치
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];

    for (int i = 1; i <= n; i++) { // 물건 1~n
        for (int j = 1; j <= k; j++) { // 배낭의 임시 용량 1~k
            // 물건 i의 무게가 배낭의 임시 용량을 초과하면
            if (w[i] > j)
                DP[i][j] = DP[i - 1][j]; // 물건 i-1까지만 담는다.
            else
                DP[i][j] = max(DP[i - 1][j], DP[i - 1][j - w[i]] + v[i]);
        }
    }

    cout << DP[n][k];

    return 0;
}
```

제출 번호	아이디	문제	결과	메모리	시간	언어	코드 길이	제출한 시간
35384471	jxlhe46	12865	맞았습니다!!	41476 KB	36 ms	C++17 / 수정	3166 B	7초 전
35384457	jxlhe46	12865	맞았습니다!!	41476 KB	40 ms	C++17 / 수정	3115 B	37초 전

참고 자료

- 양성봉, "알기쉬운 알고리즘", 생능 출판사, p.191~200.
- https://gsmesie692.tistory.com/113