

Problem Set #3 (Algorithms)

Department: 컴퓨터정보공학부

Student ID: 2018202065

Student Name: 박철준

Consider the 0-1 knapsack problem where a thief robbing a store finds n items for some integer n . The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W . Consider $0 \leq n \leq 5$, $1 \leq v_i \leq 50$, $1 \leq w_i \leq 5$, and $0 \leq W \leq 5$.

1. For a bottom-up dynamic-programming algorithm to compute the value (in dollars) of an optimal solution to the 0-1 knapsack problem for n items in $O(nW)$ time,

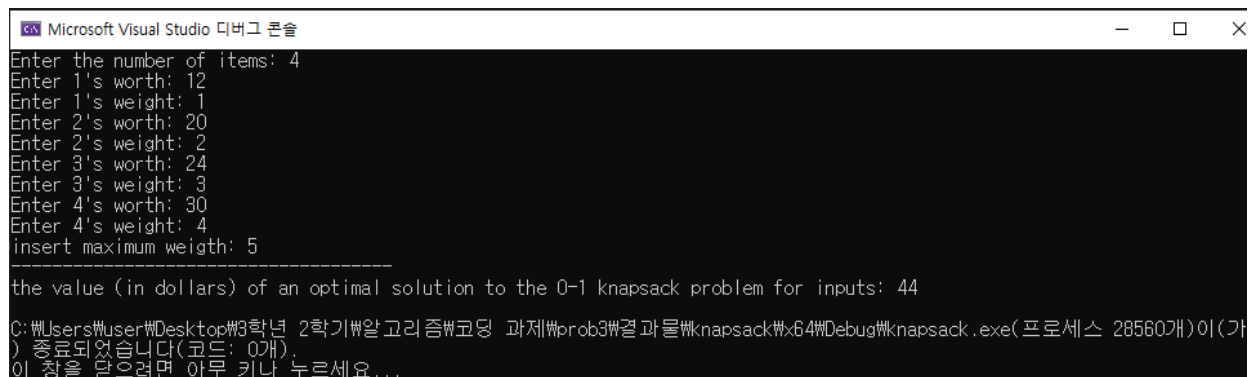
(a) Write your program that includes your comments.

코드 파일로 대체하였습니다.

(b) When $W = 5$, $v_1 = 12$, $v_2 = 20$, $v_3 = 24$, $v_4 = 30$, and $w_i = i$ for $i = 1, 2, 3, 4$, show the value (in dollars) of an optimal solution to the 0-1 knapsack problem for 4 items by executing your program.

답: 44

아래는 결과 화면입니다.



```
Microsoft Visual Studio 디버그 콘솔
Enter the number of items: 4
Enter 1's worth: 12
Enter 1's weight: 1
Enter 2's worth: 20
Enter 2's weight: 2
Enter 3's worth: 24
Enter 3's weight: 3
Enter 4's worth: 30
Enter 4's weight: 4
insert maximum weigh: 5

-----
the value (in dollars) of an optimal solution to the 0-1 knapsack problem for inputs: 44
C:\Users\User\Desktop\3학년_2학기\알고리즘\교과\과제\prob3\결과물\knapsack\x64\Debug\knapsack.exe(프로세스 28560개)이(가)
) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

(c) Explain your program and your execution.

프로그램 실행 방법 및 input 변경 방법은 다음과 같습니다.

1. Item의 개수를 입력한다. $0 \leq n \leq 5$ 범위를 벗어나면 예외 처리되어 종료된다.
2. Item의 개수만큼 차례대로 아이템의 weight과 worth를 입력한다. $1 \leq w_i \leq 5$, $1 \leq v_i \leq 50$ 의 범위를 벗어나면 예외 처리되어 종료된다.
3. 마지막으로 배낭에 담을 수 있는 무게의 한계를 입력한다. $0 \leq W \leq 5$ 의 범위를 벗어나면 예외 처리되어 종료된다.

입력을 받은 다음은 Knapsack_Dynamic_programing 함수가 실행되고 이후 메커니즘은 다음과 같습니다.

1. [짐의 수+1][무게+1] 크기의 전역 이차원 배열 변수를 사용한다. 위 문제의 경우 짐이 4개이고 가방에 5kg까지 담을 수 있으니 2차원 배열을 아래와 같이 표를 가지고 설명하고자 한다.

	1	2	3	4	5
1 (1,12)					
2 (2,20)					

3 (3,24)					
4 (4,30)					

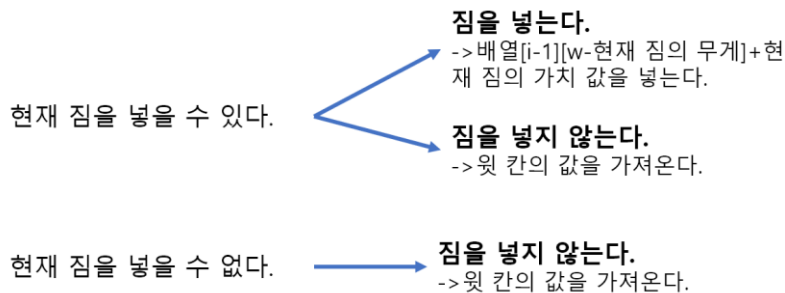
2. 이중 for문을 통해 2차원 배열 즉 표의 칸을 채워 나간다. 바깥 for문의 경우 item의 순번을 증가시키고 안쪽 for문의 경우 배낭 무게의 한계를 1씩 증가시킨다. 우선 첫 번째 행의 경우는 가방에 무게1, 가치 12의 짐을 넣으려 하는 경우이다. 담을 수 있다고 하는 경우에만 짐을 넣을 수 있다. 따라서 표는 아래와 같이 된다.

	1	2	3	4	5
1 (1,12)	12	12	12	12	12
2 (2,20)					
3 (3,24)					
4 (4,30)					

3. 두 번째 행의 경우는 가방에 무게2, 가치 20의 짐을 넣으려 하는 경우이다.

	1	2	3	4	5
1 (1,12)	12	12	12	12	12
2 (2,20)	12	20	32	32	32
3 (3,24)					
4 (4,30)					

1행 2열의 경우 가방 무게가 1이어서 담을 수 없는 경우 표 위의 값을 가져와 12가 된다. 2행 2열의 경우 가방에 담을 수 있는 경우이고 이때의 메커니즘은 표의 2-1행 2-2열 즉 0값과 현재 짐의 가치 즉 20 값을 더한 20값과 표 위 값 12를 비교하여 큰 값을 선택하여 채운다. 더 붙여 설명하고자 2행 3열의 값을 채워 넣는다고 가정하면 2-1행 3-2열 즉 12값과 현재 짐의 가치 즉 20값을 더한 32값과 표 위 값 12를 비교하여 큰 값을 선택하여 채운다. 즉 32로 채운다. 즉 우리는 여기서 알고리즘의 점화식을 구할 수 있다. 경우는 아래와 같다.



이를 수식으로 표현하면 다음과 같다.

$c[i, w]$ 를 the value of the solution for items 1 ~ i 라하고 w 를 담을 수 있는 가방의 무게의 한계라고 가정하면

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ c[i-1, w] & \text{if } i, w > 0 \text{ and } w_i > w, \\ \max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i, w > 0 \text{ and } w_i \leq w. \end{cases}$$

해당 수식으로 나타낼 수 있다.

4. 이러한 과정을 참고하여 표의 나머지 부분을 채우면 다음과 같다

	1	2	3	4	5
1 (1,12)	12	12	12	12	12
2 (2,20)	12	20	32	32	32
3 (3,24)	12	20	32	36	44
4 (4,30)	12	20	32	36	44

5. 이제 입력받은 $n = 4$ 즉 item 개수의 최대값과 $W=5$ 즉 배낭 무게의 최대 한계를 통하여 optimal value를 구할 수 있다. 저장한 2차원 배열에서 item개수행 배낭 무게열에서 입력받은 n 행 W 열 즉 4행 5열의 값이 구하려 하는 the value (in dollars) of an optimal solution to the 0-1 knapsack problem for 4 items이고 해당 값은 44이다.
6. 또한 해당 알고리즘을 코드로 나타내면 아래와 같다.

```
1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4
5  int n = 0; // 짐의 개수
6  int W = 0; // 배낭 무게의 한계
7  int v_arr[6] = {0}; // 각각의 짐의 가치를 저장할 배열
8  int w_arr[6] = {0}; // 각각의 짐의 무게를 저장할 배열
9  int profit[7][7] = {0,}; // 다이나믹 프로그래밍 과정을 저장할 2차원 배열
10
11 void Knapsack_Dynamic_programing()
12 {
13     for (int i = 1; i <= n; i++)
14     {
15         // i는 item의 순번을 나타내며 i를 1씩 증가시키며 표를 완성시키는 과정.
16         for (int j = 1; j <= W; j++)
17         {
18             //j를 가장 무게의 최대값으로 두고 j를 1씩 증가시키며 표를 완성시키는 과정
19             if (w_arr[i] <= j) //현재 짐을 넣을 수 있을 때
20             {
21                 //표의 위칸의 값과 (현재짐의 가치+표에서 i-1행의 배낭 무게 한계 - 현재 짐의 무게열의 값을 가져옴)값 중 최대값을 선택한다.
22                 profit[i][j] = max(profit[i - 1][j - w_arr[i]] + v_arr[i], profit[i - 1][j]);
23             }
24             else // 현재 짐을 넣을 수 없을 때
25             {
26                 //표의 위칸의 값을 가져온다.
27                 profit[i][j] = profit[i - 1][j];
28             }
29         }
30     }
31 }
```

위에서 설명했던 동작 방식 그대로 작성한 코드가 작성됨을 알 수 있다.