

Problem Set #3 (Algorithms)

Department: 컴퓨터정보공학부

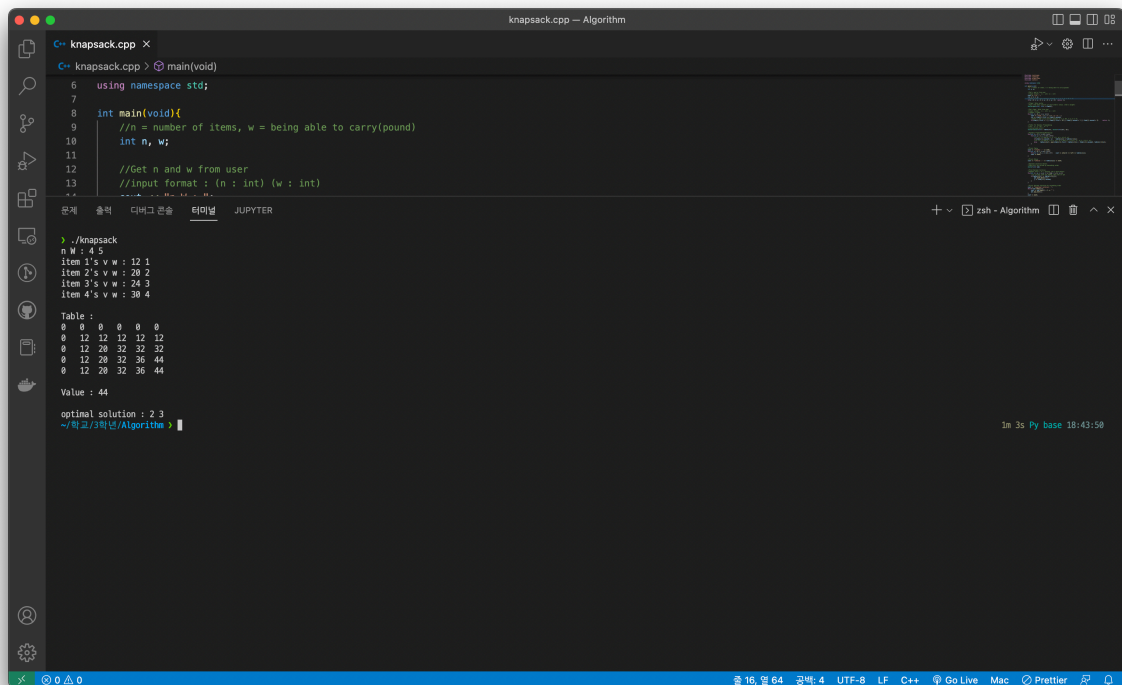
Student ID: 2018202046

Student Name: 이준휘

(a) Write your program that includes your comments.

코드 파일로 대체

(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.



```
knapsack.cpp — Algorithm
C++ knapsack.cpp > main(void)
6 using namespace std;
7
8 int main(void){
9     //n = number of items, w = being able to carry(pound)
10    int n, w;
11
12    //Get n and w from user
13    //input format : {n : int} {w : int}
14
15    //n w
16    4 5
17
18    item 1's v w : 12 1
19    item 2's v w : 20 2
20    item 3's v w : 24 3
21    item 4's v w : 30 4
22
23    Table :
24    0 0 0 0 0 0
25    0 12 12 12 12 12
26    0 12 20 32 32 32
27    0 12 20 32 36 44
28    0 12 20 32 36 44
29
30    Value : 44
31
32    optimal solution : 2 3
33    ~~~~
34    In 3s Py base 18x43x58
```

(c) Explain your program and your execution

해당 프로그램은 g++ 환경에서 컴파일 된 c++ 코드다. 동작은 다음과 같이 진행된다.

프로그램은 우선 n (item 개수)값과 w (가방 운반 가능 무게)값을 입력 받는다. 두 값은 띄어쓰기 형태로 존재하며, 위의 b 번과 같이 “4 5” 형태로 순차적으로 입력하면 된다. 만약 n 과 w 의 값이 지정된 범위에서 벗어난 값일 경우 -1을 반환하고 프로그램을 종료한다.

이후 각 item의 정보를 저장할 vector를 선언한다. item의 정보에는 v (value)와 w (weight), 두 개의 정보가 있기 때문에 vector는 $\text{pair}<\text{int}, \text{int}>$ 형태로 데이터를 관리한다. item의 개수인 n 개로 vector를 초기화한다.

vector를 선언한 후 item의 정보를 입력 받는다. 이는 이전의 $n w$ 값 입력과 같은 형식으로 “int int”

형태로 v 값과 w 값을 순차 입력 받는다. 각 입력에서 v 와 w 값이 정해진 범위를 벗어난 경우 -1을 반환하고 프로그램을 종료한다. 위의 입력 과정은 n 회 반복된다.

Dynamic Programming을 위한 `vector<vector<int>>` 형태의 table은 $(n+1) * (w+1)$ 크기의 행렬로 이루어져 있다. 행 index는 item number를 의미하며, 열 index는 weight를 의미한다. 그리고 각 table의 값은 1 ~ 해당 index의 item까지 있으며 w 가 index 위치와 같을 때의 최대 value 값을 나타낸다. 해당 table은 0으로 초기화한다.

기본적으로 Optimal Substructure of the 0-1 Knapsack Problem에서 $i = 0$, or $w = 0$ 의 조건에서는 0이기 때문에 반복문 범위에서 해당 index 이후를 초기값으로 한다.

만약 w_i (현 위치의 item 무게)가 > 현 index 위치의 w 를 벗어난 위치일 경우, 즉 $item[i-1]$ ($i-1$ 로 한 이유는 item vector에서는 item의 index가 0부터 시작하기 때문이다.)의 weight가 현 index 위치의 w 를 초과한 경우, 현재 table 위치의 값은 이전 $[i-1][j]$ 위치의 값에서 그대로 가져온다. 이는 해당 item이 전체 무게의 조건을 통과하지 못하기에 넣지 않는다는 의미다.

만약 $w_i \leq$ index 위치의 w 일 때에는 이전 위치($[i-1][j]$)의 table 값과 $[i-1][w - w_i]$ 위치(w_i 를 더하기 전 가장 높은 value)에서 현 item의 value를 더했을 때 값과 비교하여 최대값을 현재 table의 index 위치에 삽입한다.

이후 table을 출력하는 구문 “Table : “ 이후로 작성되어 있다.

(b)에서 넣은 table의 값을 보았을 때 첫 번째 item을 넣었을 때에는 item 1의 무게가 1임으로 $w = 1 \sim 5$ 위치에 모두 넣을 수 있기에 12로 값이 출력되었다. 두 번째 item을 넣었을 때에는 item 2의 무게는 2임으로 $w = 1$ 일 때에는 넣을 수 없음으로 12를 그대로 가져가며, $w = 2$ 일 때에는 해당 item을 넣지 않을 경우 w 는 0에서 가져옴으로 $0 + 20$ 과 12를 비교하여 큰 값인 20을 취하며, w 가 3 이상일 경우에는 $w = 1$ 이상일 때의 이전 위치의 value값은 12임으로 $12 + 20$ 과 12를 비교하여 큰 값인 32를 취한다. 이러한 방식으로 table을 채워가며 table의 끝 값인 44까지 출력되었다. 각 table의 값은 해당 index에서의 optimal substructure의 value 값을 의미한다.

Value를 출력하는 구문은 “Value : “ line에 작성되어 있으며, table의 가장 끝 위치의 값이 bottom-up으로 계산한 optimal solution의 value 값에 해당된다. (b)의 결과에서는 $n = 4$, $w = 5$ 일 때의 최종 optimal solution의 값이 44에 해당한다는 것이다.

이후의 과정은 추가적으로 해본 Back tracking을 이용한 optimal solution을 탐색하는 과정이다. 반복문을 통해 현 위치의 value와 이전 위치의 value를 비교하여 바뀐 경우 해당 위치의 item이 사용되었다는 것을 opt에 삽입한다. 그리고 이를 출력하는 과정이다.

(b)에서는 2와 3으로 결과가 나왔으며, 두 무게를 더했을 때 $2 + 3 = 5$ 으로 무게를 만족하며, 두 value를 더했을 때 $20 + 24 = 44$ 로 value값을 만족하는 것을 보여, 해당 알고리즘이 정상적으로 구현되었음을 검증하였다.