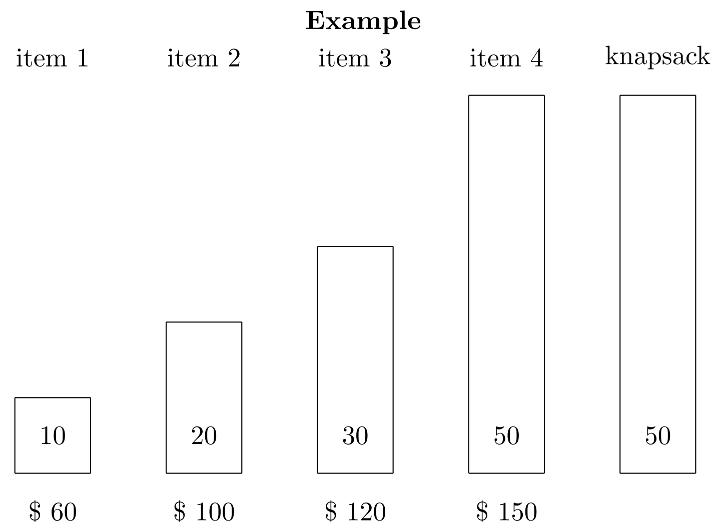
Dynamic Programming Solution to the Discrete Knapsack Problem

Cheng Li, Virgil Pavlu, Javed Aslam

Discrete Knapsack Problem

Given a set of items, labelled with $1, 2, ..., n$, each with a weight w_i and a value v_i, determine the items to include in a knapsack so that the total weight is less than or equal to a given limit W and the total value is as large as possible.



Greedy Algorithm?

Can we solve it with a greedy algorithm? Let's try three different greedy strategies for this example:

1. largest to smallest $/lb

   We pick items 1 and 2. Value=160.

2. largest to smallest value

   We pick item 4. Value=150.

3. smallest to largest weight

   We pick items 1 and 2. Value=160.

However, if we pick items 2 and 3, we get value=220. So greedy algorithms do not work. We can use dynamic programming to solve this problem.

Dynamic Programming Methodology

(1) Characterize the Structure of an Optimal Solution. The Discrete knapsack problem exhibits optimal substructure in the following manner.

Claim 1 *Let i be the highest-numbered item in an optimal solution S for W pounds and item $1..n$. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and items $1..i-1$.*

Proof: By contradiction, suppose that there was a better solution for $W - w_i$ pounds and items $1..i-1$ than S'. Then S' could be replaced with this better solution, yielding a valid solution for W pounds and item $1..n$ with larger value than the solution being considered. But this contradicts the supposed optimality of the given solution, $\rightarrow\leftarrow$. $\square$

Thus, the optimal solution to knapsack problem is composed of optimal solutions to smaller subproblems.

(2) Recursively Define the Value of the Optimal Solution. First, we define in English the quantity we shall later define recursively. Let $c[i, w]$ be the largest value for items $1..i$ and maximum weight w. Then the optimal solution for items $1..i$ either include item i, in which case it is v_i plus a subproblem solution for items $1..i-1$ and the weight excluding w_i, or doesn't include item i, in which case it is a a subproblem solution for items $1..i-1$ and the same weight. That is, if we pick item i, we take v_i value, and we can choose from items $1..i-1$ up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if we decide not to take item i, we can choose from items $i..i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made. We thus have the following recurrence.

Claim 2 $c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ c[i-1, w] & \text{if } w_i > w, \\ \max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i. \end{cases}$

Proof: The correctness of this recursive definition is embodied in the paragraph which precedes it. $\square$

There is another way to recursively define the value of the optimal solution. Suppose in the solution for items $1..i$ and maximum weight w, the highest-numbered item picked is item j. Then the optimal solution for items $1..i$ is v_j plus a subproblem solution for items $1..j-1$ and the weight excluding w_j. We don't know what the highest-numbered item picked is, so we try all possibilities for $j \leq i$. We thus have the following recurrence.

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0, \\ \max_{j \leq i : w_j \leq w} \{v_j + c[j-1, w-w_j]\} & \text{other.} \end{cases}$$

Compared with the previous recursive formula, this formula checks more possibilities, and thus introduces more computation. In fact, the time complexity of the first approach is $\Theta(nW)$, while the second is $\Theta(n^2 W)$. So we will implement the first formula.

(3) Compute the Value of the Optimal Solution Bottom-up. The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = (v_1, v_2, ..., v_n)$ and $w = (w_1, w_2, ..., w_n)$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value we can take.

```
DYNAMIC-DISCRETE-KNAPSACK(v, w, n, W)
1  for w ← 0 to W
2      do  c[0, w] ← 0
3  for i ← 1 to n
4      do  c[i, 0] ← 0
5      for w ← 1 to W
6          do if w_i ≤ w
7                then if v_i + c[i − 1, w − w_i] > c[i − 1, w]
8                        then c[i, w] ← v_i + c[i − 1, w − w_i]
9                        else c[i, w] ← c[i − 1, w]
10               else c[i, w] ← c[i − 1, w]
```

Claim 3 *When the above procedure terminates, for all $0 \leq i, w \leq n$, $c[i, w]$ will contain the largest value for items $1..i$ and maximum weight w.*

Proof: The correctness of the above procedure is based on the fact that it correctly implements the recursive definition given above. The base case is properly handled in Lines 1 to 4, and the recursive case is properly handled in Lines 3 to 10. Note that since the loop defined in Line 3 goes from 1 to n, no element of c is accessed in Lines 7-10 before it has been computed. □

(4) Construct the Optimal Solution from the Computed Information. Consider the following piece of pseudocode, where c is the matrix computed above, w is the weight sequence, n is the number of items, and W is the maximum weight.

```
ITEMS(c, w, n, W)
1  while n > 0 and W > 0
2      if c[n, W] > c[n − 1, W]
3      print n
4      W ← W − w[n]
5  n ← n − 1
```

Claim 4 *The above procedure correctly outputs an optimal set of items picked.*

Proof: The set of items to take can be deduced from the c table by starting at $c[n, W]$ and tracing where the optimal value came from. If $c[i, w] = c[i - 1, w]$, item i is not part of the solution, and we continue tracing with $c[i - 1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i - 1, w - w_i]$. □

(5) Running Time and Space Requirements. The DYNAMIC-DISCRETE-KNAPSACK procedure runs in $\Theta(nW)$ due to the nested loops (Lines 3 and 5), and it uses $\Theta(n^2)$ additional space in the form of the matrix c. The ITEM procedure runs in time $O(n)$ to trace the solution (since it starts in row n of the table and moves up 1 row at each step). It uses no additional space beyond the inputs given. Thus, the total running time is $\Theta(nW)$ and the total space requirement is $\Theta(n^2)$.