

Dynamic Programming Example: 0/1 Knapsack Problem

Note: this is another dynamic programming example to supplement those in given in lecture and the readings. This document *may* only make sense if you're studied the lecture notes and readings on dynamic programming. It is not necessarily intended to be "stand-alone."

The problem:

Input: a set of $S = \{ s_1, \dots, s_n \}$ of n items where each s_i has value v_i and weight w_i , and a knapsack capacity W .

Required solution: choose a subset O of S such that the total weight of the items chosen does not exceed W and the sum of items v_i in O is maximal with respect to any other subset that meets the constraint.

Compare to the continuous knapsack problem:

- In continuous knapsack, we're allowed to add a fraction x_i of each item to the knapsack
- This one called 0/1 knapsack because same as requiring each x_i to be either 0.0 or 1.0.
- Recall we found optimal solution for continuous knapsack when our greedy choice function picked as much as possible of the next item with the highest value-to-weight ratio.

Example Instance of this Problem:

Let $W = 4$ and

Item i	v_i	w_i	Ratio v_i / w_i
1	3	1	3
2	5	2	2.5
3	6	3	2

Will the greedy approach used on the continuous knapsack produce the optimal solution here?

Answer: no! The optimal solution is to choose items 1 and 3, with total value of 9 and total weight 4 (which fills the knapsack).

Self-test Question: What solution will be chosen by the greedy approach?

A dynamic programming solution can be designed that produces the optimal answer. To do this, we must:

1. Identify a recursive definition of how a larger solution is built from optimal results for smaller subproblems.
2. Create a table that we can build bottom-up to calculate results for subproblems and eventually solve the entire problem.

How can we break an entire problem down into subproblems in a way that uses optimal results of subproblems? First we need to make sure we have a clear idea of what a subproblem solution might look like.

Recursive Definition of Solution in terms of Subproblem Solutions:

Suppose the optimal solution for S and W is a subset, call it O , in which s_k is the “highest numbered” item in the sequence of items that makes up O . For example, the items in O might be displayed in bold in S as shown below:

$$S = \{ s_1, s_2, \mathbf{s_3}, \dots, s_{k-1}, \mathbf{s_k}, \dots, s_n \}$$

Then, $O - \{ s_k \}$ is an optimal solution for the subproblem $S_{k-1} = \{ s_1, \dots, s_{k-1} \}$ and knapsack capacity $W-w_k$. The value of the complete problem S would simply be the value calculated for this subproblem S_{k-1} plus the value v_k .

Our approach will calculate values $V[k,w]$ which represent the optimal value of subproblems $S_k = \{ s_1, \dots, s_k \}$ and any target weight w (where $0 \leq w \leq W$). Each value $V[k,w]$ represents the optimal solution to this subproblem:

- What would the value be if our knapsack weight was just w and we were only choosing among the first k items?

Assume for some given values of k and w we already had a correct solution to a subproblem stored in $V[k-1,w]$. We want to extend this subproblem, and the question at this point is now:

- Can I add item s_k to the knapsack, and if I can will this improve the total? (I might be able to add this item but only if I remove one or more items that reduces the overall value. I don't want that!)

There are really three cases to consider when calculating $V[k,w]$ for some given values of k and w :

1. Let's assume for the moment that I am able to add this item s_k to the knapsack that has capacity w . In this case, the total value for this subproblem S_k will be v_k plus the best solution that exists for the $k-1$ items that came before s_k for the smaller knapsack weight $w-w_k$. (If we're going to add the k th item, we have less room for the previous $k-1$ items.)
In this case, the value for $V[k,w]$ will thus be $v_k + V[k-1,w-w_k]$
2. But adding this item may not be a good idea. Perhaps the best solution at this point does not include this k th item. In that case, the value $V[k,w]$ would be what we had for the previous $k-1$ items, or simply $V[k-1,w]$.

- It might not be possible to add this item to the knapsack – there may not be room for it! This would be the case if $w - w_k < 0$ (that is, $w < w_k$). In this case, we can't add the item, so the value to store would be the best we had for the $k-1$ items that came before it, $V[k-1, w]$ (same as case 2 above).

At this point, we know how to calculate the value for $V[k, w]$ for given values of k and w . The pseudo-code looks like this:

```

if ( not room for Item k )
    V[k,w] = V[k-1,w] // best result for k-1 items
else if ( best to add Item k )
    V[k,w] = v_k + V[k-1, w-w_k] // Case 1 above
else // not best to add Item k
    V[k,w] = V[k-1,w] // best result for k-1 items

```

How do we know if it's best to add Item k ? We simply compare the values that would be assigned to $V[k, w]$ in the last two cases of the if/else sequence above. Thus this code fragment would become:

```

if (w-w_k < 0) // not room for item k
    V[k,w] = V[k-1,w] // best result for k-1 items
else {
    val_with_kth = v_k + V[k-1, w-w_k] // Case 1 above
    val_for_k-1 = V[k-1,w] // best result for k-1 items
    V[k,w] = max( val_with_kth, val_for_k-1 )
}

```

Using a Table to Calculate Results:

To use this information to calculate the answer we want, $V[n, W]$, we will create a table $V[0..n, 0..W]$.


- Note that the columns will represent an increasing value of the target weight w from 0 up to the knapsack capacity. Thus moving along a row to the next larger column represents asking “do we get a better answer for k items if we have a little more capacity?”
- Note that the rows represent an increasing number of items. Thus moving down to the next row while staying in the same column represents asking “can we add the next item with this same capacity w ?”

There are some obvious base cases that we can calculate directly:

- $V[0, w] = 0$ for $0 \leq w \leq W$. The value is zero if you don't choose an item.
- $V[k, 0] = 0$ for $0 \leq k \leq n$. If you have a knapsack with zero capacity, you can't add an item to it.

We can compute the table in bottom-up fashion by working in row-major fashion, i.e. calculating an entire row for increasing values of w , then moving to the next row, etc.

$V[k,w]$	$w=0$	1	2	...	W
$k=0$	0	0	0	0	0
1	0				→
2	0				→
...	0				→
n	0				→



Combining this data structure with the algorithm given above for calculating a $V[k,w]$ value results in the following pseudo-code for that solves the entire problem:

```

Knapsack(v, w, W) {
    for (w = 0 to W) V[0,w] = 0
    for (k = 0 to n) V[k,0] = 0
    for (k = 1 to n) {
        for (w = 1 to W) {
            if (w-wk< 0) // not room for item k
                V[k,w] = V[k-1,w] // best result for k-1 items
            else {
                val_with_kth = vk + V[k-1, w-wk] // Case 1 above
                val_for_k-1 = V[k-1,w] //best result for k-1 items
                V[k,w] = max( val_with_kth, val_for_k-1 )
            }
        }
    }
    return V[n,W]
}

```

Recovering the Items that Produce the Optimal Value:

The value returned by the function, $V[n,W]$, is the value of the optimal solution. But which subset of items make up O , the subset of S with the maximal value that fit in the knapsack? We will find this by tracking “backwards” through the values in the table, starting at $V[n,W]$. At each point of our trace, we can tell by the values whether or not the current item (corresponding to the current row value, v) was part of the optimal solution.

Note that when we’re building the table, the value at $V[k,w]$ will be set to be $V[k-1,w]$ for both cases where s_k is not added to the partial solution (cases 2 and 3 described above). Therefore, in our trace back through the table, if $V[k,w]$ equals $V[k-1,w]$ then s_k was not part of the optimal solution. We continue to trace at $V[k-1,w]$.

But if $V[k,w]$ does **not** equal $V[k-1,w]$, then item s_k **was** part of the solution. In this case, we continue the trace one row higher at $V[k-1,w-w_k]$, to see if the $k-1$ item was part of the solution. (Note how this corresponds to case 1 described above.)

Complexity:

Since the time to calculate each entry in the table $V[k,w]$ is constant, the time complexity is $\Theta(n \times W)$.

This is not an in-place algorithm, since the table requires $\Theta(n \times W)$ cells and this is not linear in n . But dynamic programming algorithms save time by storing results, so we wouldn't expect any dynamic programming solution to be in-place.

Important: Is the time-complexity of this solution polynomial in terms of its input sizes? It might appear to be so, but technically this is an exponential solution. Why?

Consider the size of the input values that make up the values and weights. There are two values for each of the input items.

But, consider the value W , the knapsack capacity. This is one value, and it's always one input item no matter what value it takes on. But if this value changes, the size of the table and the time it takes to create it changes. For example, if the capacity doubles, then the execution time and the space used also double.

But is this different than, say, sequential search? In that problem, if n is the number of items in the array, if n doubles then the time of execution also doubles (since sequential search has linear time-complexity). But n is a count of the input items in this problem, whereas in the knapsack problem, the capacity W is a value that is processed (not a count of input items). The complexity depends on the size of this single value.

So as noted in the discussion of NP and NP-complete problems, the issue of encoding of inputs must be taken into account for the knapsack problem. What is the size of a single value W ? How it is encoded?

Most often we think of integer values as being encoded in binary notation. The size of a value is the number of bits required to store it. Thus if the size of an input storing a value like W increases by one (i.e., one bit), then that input could represent twice as many values.

For this reason, the complexity of the dynamic programming solution for the knapsack problem (and many other problems) grows exponentially. For this problem, if the size of W increases by one bit, the amount of work doubles. Compare this to a problem where the amount of work is proportion to 2^n : if the input size increases to $n+1$, then the amount of work doubles.

However, as is true for many algorithms with exponential time-complexity, this solution will run in a reasonable amount of time for many values of W and n .