A Course in Graduate Algorithms **Lecture 4.1**

Dynamic Programming - The Knapsack Problem

Designed by Prof. Bo Waggoner for the University of Colorado-Boulder *Updated: 2023*

In this problem, we are given a set of items $i = 1, \dots, n$ each with a *value* $v_i \in \mathbb{R}_+$ (a positive number) and a *weight* or size $w_i \in \mathbb{N}$ (a nonnegative integer).

We are given a number $W \in \mathbb{N}$ which is the maximum weight our knapsack can hold, also called the capacity or size of the knapsack. We must find the max-value subset of items that can fit in the knapsack. There are several versions of this problem; let's look at the first.

1 Duplicates allowed

In this version of the knapsack problem, there are unlimited copies of each item available (energy bars, etc.).

Recall the elements of a DP solution: subproblem definition, computing the final value, the recurrence, and reconstructing the solution. Here a natural **subproblem** is to have a smaller-capacity knapsack. Namely, let $C[w]$ = the maximum value we can fit in a knapsack of size w. With this subproblem, **computing the final value** is easy, as it is just $C[W]$.

For the **recurrence**, as usual, we need a mathematical structure or fact. For the **base case** where $w = 0$, i.e. no items can fit, the value is zero. So we set $C[0] = 0$. For the **inductive case**:

Claim 1. *For $w \geq 1$,*

$$C[w] = \max \begin{cases} C[w-1] \\ \max_{i:w_i \leq w} \; v_i + C[w - w_i] \end{cases}.$$

In other words, we can write this as an algorithm:

- Set $C[w] = C[w-1]$, in other words, consider the optimal solution for a knapsack of size $w-1$. That solution can fit in this knapsack as well, since this one is only larger.
- For each item i:
  - Check if item i can fit in this knapsack, i.e. check if $w_i \leq w$. If not, skip this item and keep going.
  - Put item i in the knapsack. We now have a space of $w - w_i$ remaining, and we have a value of v_i.
  - Fill the remaining space optimally. Luckily, we already solved that subproblem: it gives a value of $C[w - w_i]$.
  - Check if the resulting total value, $v_i + C[w - w_i]$, is better than our current solution. If so, keep it.

Now let's prove the recurrence correct:

Proof of Claim 1. Given w, either the optimal solution is empty or it has at least one item. If it's empty (i.e. no items fit in the knapsack), then we can just set $C[w] = C[w-1]$. Acutally, both will be zero in this case.

Suppose the optimal solution is not empty. Then it has at least one item, say i. Now for the remaining space $w - w_i$ (which is at least zero since i fits in the knapsack), it must be used optimally. So the total value from the remaining space is $C[w - w_i]$. (If it were not used optimally, then we could get a better solution for the space and then add item i to it and obtain a better solution for $C[w]$, which contradicts the assumption that this is optimal.)

So $C[w] = v_i + C[w - w_i]$. So the recurrence is correct, since the optimal solution is the result of picking the best such item i. □

Combining these gives a dynamic programming algorithm:

Algorithm 1 Knapsack (duplicates allowed)

```
Input: Item values v_1, ..., v_n and sizes w_1, ..., w_n; capacity W
Output: optimal total value in the knapsack
Set C[0] = 0
for w = 1, ..., W do
   Set C[w] = C[w − 1]
   for i = 1, ..., n do
      if w_i ≤ w then
         Set C[w] = max{C[w] , v_i + C[w − w_i]}.
      end if
   end for
end for
Return C[W]
```

Correctness: As usual in dynamic programming, correctness follows from correctness of the DP elements, which were argued above.

Efficiency: Space usage is dominated by C, which uses $O(W)$ space. For running time, we have nested loops, the outer one has W iterations and the inner one has n iterations, and the interior operations are constant time per iteration. So running time is $O(nW)$.

Exercise 1. Would you consider this algorithm efficient?

Exercise 2. More formally, is it officially polynomial time in the input size? *Hint:* if I multiply all the sizes by 100 and the capacity by 100, the problem doesn't change and the input size doesn't change (as long as it fits in the RAM model's word size). What happens to running time?

Exercise 3. Modify the algorithm to reconstruct the actual list of items in the optimal knapsack. *Hint:* Recall that for reconstruction, we should keep track of the choices our algorithm needed to make at each subproblem. At subproblem $C[j]$, what were our choices? Then, how do we backtrack from the very end, i.e. $C[W]$, to the beginning?

The knapsack algorithm runs in what's called *pseudopolynomial* time: polynomial in the length of the input and the size of the largest integer in the input.

2 No Duplicates

In this version of the problem, there is just one copy of each item.

We might hope to modify the previous solution while keeping the subproblem $C[w]$ essentially the same. For example, by somehow remembering which items were used in $C[w]$. This turns out to fail, in part because there could be multiple optimal subsets for $C[w]$, and remembering all of them turns out to be prohibitive. Instead, we need a trick similar to FLoyd-Warshall.

Subproblem. The solution is to **introduce an extra dimension.** Specifically, let $C[k, w]$ be the maximum value one can obtain from a knapsack of size w using only items from the subset $\{1, \ldots, k\}$.

Computing the final solution. We will simply return $C[n, W]$ where n is the number of items and W is the knapsack capacity.

Recurrence. The **base case** is pretty straightforward: $C[k, 0] = 0$ for all item indexes k and $C[0, w] = 0$ for all capacities w.

For the **inductive case**, we need a fact.

Claim 2. *For $k \geq 1, w \geq 1$, we have*

$$C[k, w] = \max \begin{cases} C[k-1, w] \\ v_k + C[k-1, w-w_k] \quad \textit{(if } w_k \leq w\textit{)} \end{cases}.$$

Proof. For the optimal solution with items $1, \ldots, k$ and capacity w, there are two possibilities: we either include item k, or we don't. If we don't, then the optimal solution uses only items $1, \ldots, k-1$, so its value is $C[k-1, w]$.

If we do, then the remaining space is $w - w_k$, and to fill it, we are only allowed to use items $1, \ldots, k-1$ because we just used item k. So the remaining value is $C[k-1, w-w_k]$, and our total value is $v_k + C[k-1, w-w_k]$. Note this is only possible if $w_k \leq w$, as otherwise item k cannot fit.

Since these are the only two possibilities (or only one possibility if $w_k > w$), and the recurrence chooses the best of both, it is optimal. □

Algorithm 2 Knapsack (no duplicates)

```
Input: Item values v_1, ..., v_n and sizes w_1, ..., w_n; capacity W
Output: optimal total value in the knapsack
Set C[0, w] = 0 for all w = 0, ..., W
for i = 1, ..., n do
    for w = 1, ... W do
        Set C[i, w] = C[i − 1, w]
        if w_i ≤ w and v_i + C[i − 1, w − w_i] > C[i, w] then
            Set C[i, w] = v_i + C[i − 1, w − w_i]
        end if
    end for
end for
Return C[n, W]
```

Correctness. As usual for dynamic programming, correctness follows almost immediately from the above arguments that the three components (subproblem, final solution, recurrence) are correct.

Efficiency. Initialization takes $O(n + W)$ time, and returning the final result is constant time. There are nested loops of n and W iterations, with constant-time operations in the innermost loop, so runtime is $O(nW)$. Space is dominated by C, which uses $O(nW)$ space.

Exercise 4. How do we modify the single-use knapsack algorithm to use $O(W)$ space instead of $O(nW)$? *Hint:* the approach is somewhat similar to that of e.g. edit distance.

Exercise 5. How do we modify the single-use knapsack algorithm to return the optimal subset of items (i.e. **reconstruct** the solution, not just its value)?