DS 320 Algorithms for Data Science **Lecture #14**
Spring 2024 Prof. Kira Goldner

Dynamic Programming II: Knapsack

The Problem

Imagine that you are on a hike and you find a cave filled with riches: n riches to be exact.

Each item i in the cave has some value v_i. But it also has a weight w_i to it, and your hiking pack (or *knapsack*) can only hold up to a total weight capacity C.

Our goal is to pick which items S to take to *maximize the value* $\sum_{i \in S} v_i$ in your pack, but ensure that the weight doesn't exceed the maximum allotted, $\sum_{i \in S} w_i \leq C$.

Making the Key Observation

What key observation can we make that will help us move toward our subproblem and recurrence? A reminder of our other key observations:

Scheduling: Either the last job is in the solution, or it isn't. (Then what does that mean for the rest of the schedule and the maximum weight?)

Least segmented squares: The last point p_n belongs to a single segment which must begin somewhere. Where does it begin? In each case, what does the minimal error and optimal solution look like?

Now for knapsack:

Either we take the last item or we don't.

- If not, the optimal solution is the same as on one fewer item with the same weight capacity.
- If we *do*, the optimal solution is the same as on one fewer item with the weight capacity reduced by the weight of the last item.

But what if the last item doesn't fit? Then we definitely can't take it.

Step 1: The Subproblem

Let $\text{OPT}(i, D)$ denote the maximum total value from items $1, \ldots, i$ using total capacity at most D.

Step 2: The Recurrence

$$\text{OPT}(i, D) = \begin{cases} \text{OPT}(i-1, D) & \text{if } w_i > D \\ \max(\text{OPT}(i-1, D), v_i + \text{OPT}(i-1, D-w_i)) & \text{otherwise} \end{cases}$$

Step 3: Prove that your recurrence is correct. In the optimal solution on i items with capacity D, there are two cases: either item i is included in S or it is not. However, if w_i exceeds the capacity D, clearly i cannot be included.

Case 1: If i is otherwise not in the optimal solution, then the optimal solution is simply the optimal solution on $i-1$ items with the same capacity.

Case 2: If i is in the optimal solution, then we have the constraint that $\sum_{j\in S} w_j = \sum_{j\in S\setminus\{i\}} w_j + w_i \leq D$. Over all subsets of $\{1,\ldots,i-1\}$ that meet this feasibility constraint, $S' = \textsc{opt}(i-1, D-w_i)$ is the set that maximizes the quantity $\sum_{j\in S'} v_i$, and hence $S'\cup\{i\}$ both satisfies $\sum_{j\in S'} w_j + w_i \leq D$ and of such sets, maximizes $\sum_{j\in S'} v_j + v_i$, making it $\textsc{opt}(i, D)$.

The optimal solution between the two cases will be the one with more weight, hence the recurrence, taking the maximum of these two, is correct.

Step 4: State and prove your base cases. $\textsc{opt}(0, D) = 0$ for all D.

Step 5: State how to solve the original problem. $\textsc{opt}(n, C)$.

Step 6: The Algorithm

Algorithm 1 Knapsack($v_1,\ldots,v_n$; $w_1,\ldots,w_n$, C)

```
Input: Values v_i and weights w_i for i from 1 to n and capacity C.
Initialize memo array M of size n + 1 by C + 1
for D from 0 to C do
    M[0][D] = 0
end for
for all i from 1 to n do
    for D from 0 to C do
        if w_i > D then
            M[i][D] = M[i − 1][D]
        else
            M[i][D] = max{   M[i − 1][D],    v_i + M[i − 1][D − w_i]   }
        end if
    end for
end for
return M[n][C]
```

Step 7: Running Time

a. Pre-processing: computing base cases, sorting, etc. $O(n)$.

b. Filling in memo: This can be further broken down into

(a) Number of entries of your memo table. $O(nC)$.

(b) Time to fill each entry. Be careful of things like taking maxes over n elements! $O(1)$.

c. Postprocessing: Return statement, etc. $O(1)$.

There are $O(n)$ base cases, and a memo table of size $O(nC)$ where each entry takes $O(1)$ time to compute, so the total runtime is $O(nC)$.

Is this polynomial? If the capacity C is polynomial in the number of items n, then yes.

If $C = 2^{150}$, the C only take 150 bits to write down, but it's not about how many bits it takes to write down. The exponential size of C (and thus input to the algorithm) implies this is not at all tractable.

Definition 1. An algorithm is *pseudopolynomial* time if its runtime is polynomial in the numerical values of the inputs, but not the number of bits required to express them.

Whether knapsack is polynomial or not completely depends on C.