CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 10

February 17, 2022

Instructor: Sepehr Assadi

1 Dynamic Programming I: Knapsack

We now consider a canonical example of the application of dynamic programming: the so-called knapsack problem defined as follows:

Problem 1. The knapsack problem is defined as follows:

- **Input:** A collection of n items where item i has a positive integer weight w_i and value v_i plus a knapsack of size W.
- **Output:** The maximum value we can obtain by picking a subset S of the items such that the total weight of the items in S is at most W, i.e.,

$$\max_{S \subseteq \{1,\ldots,n\}} \sum_{i \in S} v_i$$
$$\text{subject to} \quad \sum_{i \in S} w_i \leq W.$$

We will design a dynamic programming algorithm for this problem. As we stated before, the *main step* in doing so is to write a *recursive formula/algorithm* for the whole problem in terms of the answers to smaller subproblems. This is done in two steps:

(a) **Specification:** *Describe* the problem that you are designing the recursive formula for, *in coherent and precise English*. In this step, you are *not* writing *how* to solve that problem, but *what* is the problem you are trying to solve. At this point, you should also specify how the answer to the original question can be obtained *if* we have solution for this specification.

(b) **Solution:** Give a *recursive* formula or algorithm for the problem you described in the previous step by solving the *smaller instances* of the *same exact* problem. Remember that we *always* need to prove this recursive formula indeed computes the specification we described earlier for the problem.

Again, to emphasize, once we have the above, our task is almost done: the rest can all be done in a *mechanical* way by using the memoization technique or dynamic programming as we already saw for Fibonacci numbers (and will see shortly for the knapsack problem).

We now apply this method to the knapsack problem.

(a) **Specification:** For any integers $0 \leq i \leq n$ and $0 \leq j \leq W$, define:

- $K(i, j)$: the maximum value we can obtain by picking a subset of the first i items, i.e., items $\{1, \ldots, i\}$, when we have a knapsack of size j.

Note that at this step, we are only specifying, in plain English, the problem we are attempting to solve, *not* the way we are going to solve it. We should also specify one more thing: how does solving this problem can help us in answering the original question? By returning $K(n, W)$, we can solve the original problem. This is because $K(n, W)$ is the maximum value we can obtain by picking a subset of the first n items (hence all items), when we have a knapsack of size W (the original size of knapsack).

(b) **Solution:** We now write a recursive formula for $K(i, j)$ as follows:

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1,j-w_i)+v_i, K(i-1,j)\} & \text{otherwise} \end{cases}.$$

Are we done then? Of course *not*! At this point, we have simply written some recursive formula for $K(i, j)$ but in no way it is clear that whether this formula does what it is supposed to do or not. We thus need to also prove that this is indeed a correct formula for computing $K(i, j)$.

Let us consider the base case of this function first: either when $i = 0$ or $j = 0$. In both cases, we have $K(i, j) = 0$ which is also the value we can achieve by using the first 0 items (i.e., no item at all or $i = 0$) or when the knapsack has no size (i.e., $j = 0$). So the base case of this function matches the specification.

We now consider the larger values of i and j. Suppose first $w_i > j$. In this case, $K(i, j) = K(i - 1, j)$. This is correct because we cannot fit item i in a knapsack of size j and thus the best value we can achieve is by picking the best combination of items from the first $i - 1$ items, which is captured by $K(i - 1, j)$. Thus, whenever $w_i > j$, $K(i, j) = K(i - 1, j)$ precisely captures the specification we had.

Finally, we have the case when $w_i \leq j$ (corresponding to the last line of the recursive formula). At this point, we have two options in front of us for maximizing the value of items:

(1) We either pick item i in our solution which leaves us with the first $i-1$ items remaining to choose from next and a knapsack of size $j - w_i$ but we also collected the value v_i. Hence, in this case, we can obtain the value of $K(i - 1, j - w_i) + v_i$ (remember that $K(i - 1, j - w_i)$ is the largest value we can get by picking a subset of the first $i - 1$ items in a knapsack of size $j - w_i$ which is precisely the size of our knapsack after we pick item i in the solution).

(2) The other option would be to not pick i in our solution which leaves us with the first $i - 1$ items remaining to choose from and a knapsack of the same size j (we do not collect any value in this case). This is captured by the value $K(i - 1, j)$.

But which of options (1) or (2) we should choose? Since our goal is to pick the *maximum* value we can get by picking a subset of first i items in a knapsack of size j (the definition of $K(i, j)$), we should also pick the option between (1) and (2) which gives us the *maximum value*; hence, $K(i, j) = \max\{K(i - 1, j - w_i) + v_i, K(i - 1, j)\}$ as computed by the function as well.

Are we done now? Absolutely yes! We proved that $K(i, j)$ as described by the recursive formula exactly matches the specification we provided before.

Before we move on, let us briefly mention that the proof above is an "induction in disguise": the induction hypothesis is that $K(i, j)$ matches the specification we provided and we simply followed an induction base proof followed by an induction step proof. However, we really do *not* need to each time explicitly call this an induction proof because in these scenarios, the induction hypothesis is always that the recursive function we wrote matches the specification we provided and the rest of the proof is proving this anyway without explicitly mentioning it. So, to emphasize again, in order to prove the correctness of the recursive formula, we can simply explain the its logic and show that why this matches the specification we had in both the base case and the other remaining cases.

So we are done with the main step of the problem in designing a recursive formula for our problem. We can now turn this into a dynamic programming easily.

Memoization. Memoization refers to the following very basic approach. Take the recursive formula designed in the first step and turn it into a recursive algorithm (you may actually simply design a recursive algorithm in the first place but working with a formula might be easier notationally –conceptually, the two approaches are entirely equivalent). This recursive algorithm will call itself many times when solving the subproblems and so as it is, it may repeatedly compute the same value again and again. Simply fix this by using another table to store the value of each recursive call once we computed it once and from now on, instead of repeatedly computing the value from scratch, consult the table and return the answer.

Once we do this, the runtime of the algorithm would become proportional to the *total sum* of the time spent for solving each subproblem *ignoring* the recursive calls in the subproblem (we already saw this before when we worked with recursion trees for divide and conquer recurrences; this is the same exact principle).

Specifically for the knapsack problem: We store a two-dimensional table $D[1:n][1:W]$ initialized with 'undefined' everywhere and use the following memoization algorithm: (note that we are not giving the parameters n, W as well as the arrays of weights and values to the function and instead think of them as some global parameters)

```
MemKnapsack($i, j$):

  1. If $i = 0$ or $j = 0$, return 0.
  2. If $D[i][j] \neq$ 'undefined', return $D[i][j]$.
  3. Otherwise, if $w_i > j$, let $D[i][j] =$ MemKnapsack($i-1, j$);
  4. Else, let $D[i][j] = \max\{$MemKnapsack($i-1, j-w_i$)$+ v_i$, MemKnapsack($i-1, j$)$\}$.
  5. Return $D[i][j]$.
```

It is immediate to verify that `MemKnapsack`$(i, j) = K(i, j)$ (the recursive formula we defined above) for all valid choices of i, j. Hence, the final solution to our problem is to simply return `MemKnapsack`(n, W) $(= K(n, W)$ which we said earlier is the value we are interested in). There is nothing left to prove for the correctness of this algorithm: we already did it when proving $K(i, j)$ indeed matches its description which is the maximum value we can get by picking a subset of the first i items in a knapsack of size j.

What about the runtime of this algorithm? There are n choices for i and W choices for j so there are in total $n \cdot W$ subproblems. Each subproblem also, ignoring the time it takes to do the inner recursions, takes $O(1)$ time. Hence, the runtime of the algorithm is $O(nW)$.

At this point, we are done with our dynamic programming solution for the knapsack problem and it runs in $O(nW)$ time (again, remember that memoization is a form of dynamic programming). In the following, we show how to obtain the bottom-up dynamic programming solution as well but that is *not* necessary.

Bottom-Up Dynamic Programming. In the bottom-up dynamic programming (typically called just dynamic programming), instead of relying on recursion in memoization to compute all values of our recursive formula (and store it in a table), we simply fill up the values in the table ourself. However, since we no longer rely on recursion as in memoization, we have to be *more careful*: there two additional simple yet crucial steps here that we need to take:

(c) **Identify Dependencies:** Except for the base cases, every subproblem depends on the value of other subproblems: we need to first determine for each subproblem, which other subproblems it depends on?

Specifically for the knapsack problem: Each $K(i, j)$ only depends on $K(i-1, j')$ for $j' < j$. Note that we could even determine exactly what the value of j' should be, but it is not needed for our purpose as will be evident from the next step.

(d) **Find a good evaluation order:** Now that we identified the dependencies, we need to find an order for computing the values of our recursive formula (a.k.a. entries of the table) such that in this ordering the value for each subproblem is computed *after* we first determined the values for each of the subproblems that it depends on. After this, we can simply iterate over the subproblems in this order and compute each one using the values from previously computed subproblems.

Specifically for the knapsack problem: As we argued, each subproblem $K(i, j)$ only depends on the subproblems $K(i-1, j')$ for $j' < j$. So, we simply need to compute smaller values of i and j first and work our way to the larger values. This way, for every i, j, whenever we want to evaluate $K(i, j)$, the subproblems $K(i-1, j')$ for $j' < j$ that $K(i, j)$ depends on are already computed.

Once we are done with the above two steps, getting a bottom-up dynamic programming solution for our problem is straightforward: we simply pick a table of size exactly equal to the number of subproblems (with a one to one mapping between entries of the table and the subproblems); we then iterate over the subproblems following the evaluation order we computed in the previous step and update the entries of the table (a.k.a. values of the subproblems) using the recursive formula we devised earlier. We should *not* forget to fill out the entries for base cases *first*.

Specifically for the knapsack problem: The bottom-up dynamic programming approach is as follows:

`DynamicKnapsack`:

```
1. Let $D[0:n][0:W]$ be a two-dimensional array filled with 0s (we index the table starting from 0 here
   for simplicity of notation).
2. For $i = 1$ to $n$:
   (a) For $j = 1$ to $W$:
       i. If $w_i > j$, let $D[i][j] = D[i-1][j]$.
       ii. Else, let $D[i][j] = \max\{D[i-1][j-w_i] + v_i, D[i-1][j]\}$.
3. Return $D[n][W]$.
```

Since we are following a valid evaluation order, we can show that $D[i][j] = K[i][j]$ for all i, j: this is true for the base cases and afterward, whenever we compute $D[i][j]$ from any previous entry $D[i'][j']$, we know $D[i'][j'] = K(i', j')$ at that point and thus we will also have $D[i][j] = K(i, j)$ (why did we know $D[i'][j'] = K(i', j')$? because we followed a correct evaluation order! By the way, this is again an "induction proof in disguise"). Finally, it is very easy to verify that the runtime of this algorithms is $O(nW)$ simply because its main parts are two for-loops with outer-loop iterating n times and inner-loop iterating W times and we spend $O(1)$ time per each iteration.

One Final Reminder: Dynamic programming is *not at all* about blindly filling out some tables. If you cannot *clearly* identify the subproblems you are solving and specify them in both *plain English* and have a *recursive* solution for them (as we pointed out in steps (a) and (b)), you are doing something *wrong* (very wrong!). As such, every single dynamic programming algorithm that you design in this course (and hopefully in your life!) should *always* come first with a clear specification of the recursive formula. The remaining part in designing either a memoization or a bottom-up dynamic programming approach is almost mechanical and typically up to you entirely to choose which approach you like more.

Detour: why not a greedy solution for knapsack? There seems to be greedy solution for the Knapsack problem also: sort the items based on the value v_i/w_i (i.e., the ratio of value over weight) and pick the items according to this order until we fill up the knapsack. Let us consider the following example that shows this algorithm does *not* necessarily finds the correct solution every time.

Suppose we have $n = 4$ items with weights $5, 5, 4, 4$ and values $5, 5, 3, 3$ and the total knapsack size of $W = 8$. The optimum solution for this instance is to pick the last two items which both fit the knapsack (as their total weight is $4 + 4 = 8 = W$) and have value $3 + 3 = 6$. On the other hand, if we sort the items based on their v_i/w_i values, the first two items have this ratio equal to 1 while the last two have ratio $3/4$; hence, we place either item 1 or 2 in the knapsack and at this point no other item can fit the knapsack. The returned solution would thus have value $5 < 6$ which means the algorithm does *not* find the optimum solution.

In general, *greedy algorithms rarely work correctly* (although there are cases that they do and we will study them later in the course).

2 Memoization vs Bottom-Up Dynamic Programming

Memoization is a *top-down* approach: we start by attempting to compute the value of the solution to our problem (in case of knapsack `MemKnapsack`(n, W)) and let the recursion do its job: it basically the computes the value of inner subproblems and recursively go from top (the value we are interested in) all the way to the bottom (the base cases of the recursive formula/algorithm) to compute the final answer. See Figure 1 for an illustration.

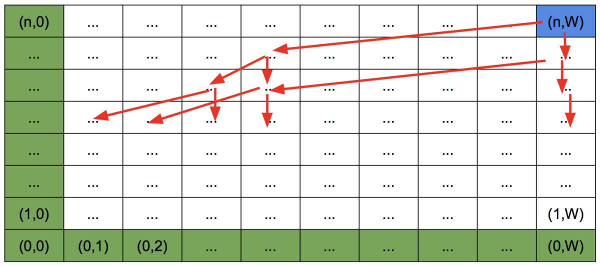

Figure 1: An illustration of the recursive calls in the memoization approach for the knapsack problem. We start at the top right (blue) cell corresponding to $K(n, W)$ (our solution) and then follow the red pointers for each recursive call until we reach the left most or bottom most (green) cells which corresponds to the base cases. Note that this figure only contains some parts of the recursive calls and not all of them.

Because in memoization we let the recursion do its job of filling the table also, it typically results in a haphazard way of filling out the table – in fact, some entries may not even be filled out if we never need their values for computing the final solution. In other words, the access to the table may not follow any obvious pattern (although there is certainly a pattern dictated by the recursive formula).

Bottom-up dynamic programming is on the other hand a *bottom-up* approach(!): we start by filling out the entries for bases cases (in case of knapsack $D[0][j]$ and $D[i][0]$) and simply follow the evaluation order we developed to fill out each entry from the previous ones. This way, we build up our solution from bottom (bases cases) all the way to the top (the value we are interested in). See Figure 2 for an illustration.

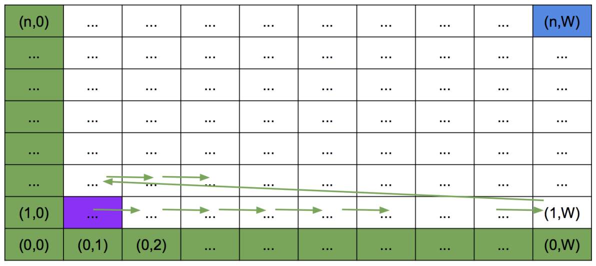

Figure 2: An illustration of the bottom-up dynamic programming approach for the knapsack problem. We start at the (purple) cell $(1, 1)$ at the bottom and work our way toward the top right most (blue) cell which is the solution we want, by using the order prescribed by green edges.

Because in bottom-up dynamic programming, we are *deliberately* filling out the tables (a.k.a. solving the subproblems), the pattern we access the table is very straightforward (we defined it ourself when finding the evaluation order). Moreover, we typically (if not always) end up solving all the subproblems in the table even though some of them may not even be necessary for computing the final answer.

We continue with another example in the next lecture.