CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 9

October 3, 2019

Instructor: Sepehr Assadi

1 Dynamic Programming and the Knapsack Problem: Recap

Let us remember the knapsack problem from the previous lecture:

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 designed a dynamic programming algorithm for this problem in the last lecture (in fact two algorithms, one using memoization, and another using bottom-up dynamic programming). The *most important step* in designing this, as well as *all* other dynamic programming solutions, is to write a recursive formula/algorithm for the problem in terms of the answers to smaller subproblems. The two steps in doing so were:

(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.

Specifically for the knapsack problem: 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.

We obtain the final solution by returning $K(n, W)$.

(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.

Specifically for the knapsack problem: We wrote a recursive formula for $K(i, j)$ as follows:

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

Remember that we were *not* done after writing this recursive formula. We should *always* (always!) prove the correctness of our recursive formula for the problem as well. However, we already did this in the previous lecture notes and thus will not repeat it again.

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 bottom-up dynamic programming as we already saw. Let us explain these parts in more detail also.

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 $D[0 : n][0 : W]$ initialized with 'undefined' everywhere (again we index the table starting from 0 not 1) 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 $D[i][j] \neq$ 'undefined', return $D[i][j]$.
  2. If $i = 0$ or $j = 0$, let $D[i][j] = 0$.
  3. Otherwise, if $w_i > j$, let $D[i][j] = \texttt{MemKnapsack}(i-1, j)$;
  4. Else, let $D[i][j] = \max\{\texttt{MemKnapsack}(i-1, j-w_i) + v_i, \texttt{MemKnapsack}(i-1, j)\}$.
  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 $O(nW)$.

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).
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.

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.

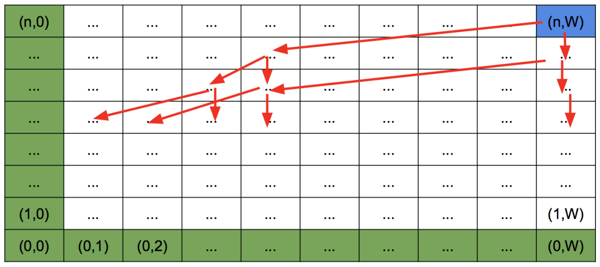

Figure 1: An illustration of the recursive calls in the memoization approach for the knapsack problem. We start at the top right (blue) cell corresponding to $K(n, W)$ (our solution) and then follow the red pointers for each recursive call until we reach the left most or bottom most (green) cells which 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 (or at least sometimes) 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.

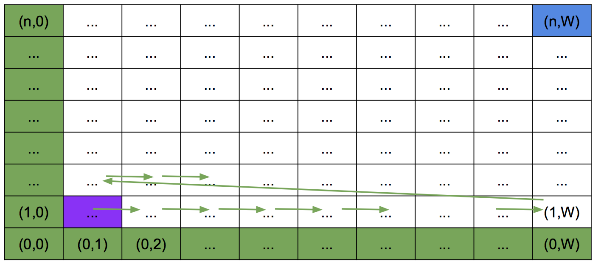

Figure 2: An illustration of the bottom-up dynamic programming approach for the knapsack problem. We start at the (purple) cell $(1, 1)$ at the bottom and work our way toward the top right most (blue) cell which is the solution we want, by using the order prescribed by green edges.

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.

3 The Longest Increasing Subsequence Problem

We now consider another canonical dynamic programming problem: the longest increasing subsequence problem. We first need a definition. For an array $A[1 : n]$, a *subsequence* of this array is any array B which is obtained by removing *zero or more* entries of A and keeping the order of the remaining elements intact. For instance, for the array $A = [1, 5, 2, 7, 6]$:

- $B = [1, 2, 7]$ is a subsequence: we removed 5 and 6 and kept the order of remaining the same;
- $B = [5]$ is a subsequence: we remove every element except for 5;
- $B = [1, 5, 2, 7, 6]$ is a subsequence: we removed *zero* entries in this case which is allowed;
- $B = [1, 2, 5, 6]$ is *not* a subsequence: order of 2 and 5 are changed from the original array;
- $B = [1, 3, 2, 7, 6]$ is *not* a subsequence: we introduced a new entry 3 out of nowhere.

We are now ready to define our problem.

Problem 2. The longest increasing subsequence problem (LIS for short) is defined as follows:

- **Input:** An input array $A[1 : n]$ of distinct integers of length n.
- **Output:** The length of the longest *increasing* subsequence of array A, i.e., the length of longest B such that (1) B is a subsequence of A, and (2) B is increasing, i.e., $B[1] < B[2] < B[3] < \cdots$.

Example: Let us consider a couple of (input, output) pairs:

- $A = [1, 2, 3, 4]$ – the answer is 4 by taking $B = A$;
- $A = [1, 3, 2, 1, 7, 4]$ – the answer is 3; there are multiple ways to achieve this, e.g., by setting $B = [1, 2, 7]$.
- $A = [4, 3, 2, 1]$ – the answer is 1; again multiple ways (4 precisely) by taking any single entry.
- $A = [1, 7, 3, 5, 4, 6, 2, 9]$ – the answer is 5; one way to achieve this is by setting $B = [1, 3, 5, 6, 9]$.

We now design a dynamic programming algorithm for this problem. The first and most important step (for the n-th time) is to write a recursive formula/algorithm for the problem, specify it in plain English, and write a solution for it (including the proof of correctness).

- *Specification (in plain English):* For every $0 \leq i \leq n$, we define:
  - $LIS(i)$: the length of the longest increasing subsequence of the array $A[1 : i]$ $\underline{\text{that ends in } A[i]}$ (we will shortly see why we need this extra part). We define $A[1 : i]$ for $i = 0$ to be the 'empty' array as a convention and thus $LIS(0) = 0$.

  The solution to our original problem is then $\max\{LIS(j) \mid 1 \leq j \leq n\}$: the actual longest increasing sequence should end in some entry of the array, say $A[k]$, and thus $LIS(k)$ would be equal to its length; by taking maximum over all choices of j, we ensure that we get the correct answer then[1].

- *Solution:* We write a recursive formula for $LIS(i)$ as follows:

$$LIS(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} \max\{1 + LIS(j) \mid 1 \leq j < i \text{ and } A[j] < A[i]\} \\ 1 \end{cases} & \text{otherwise} \end{cases}$$

  So far, we effectively wrote a formula out of nowhere. We now prove (and provide explanation) why this recursive formula captures what we specified for $LIS(i)$.

  Proof of Correctness: We consider each case of the recursive formula separately:

[1]A simple 'hack' here to avoid taking max is as follows: add one more entry to A in position $n + 1$ with value $+\infty$; then return $LIS(n + 1) - 1$ as the correct answer (instead of taking max as above): this works because we are sure that the longest increasing subsequence of this new A always ends in $A[n + 1]$.

– If $i = 0$: there is no subsequence in the empty array and hence $LIS(0) = 0$ is the correct choice.

– We now consider larger values of i. Note that firstly, by definition of $LIS(i)$ being the length of the longest increasing subsequence *that ends* in $A[i]$, we have no choice except for picking up $A[i]$ in the solution (the second (bottom) term in the outer max-term above is simply to ensure that we can always pick $A[i]$ on its own even if no elements before $A[i]$ is smaller than $A[i]$ (so the inner max-term has no value)).

  After we pick $A[i]$, the previous entries in the subsequence should all be smaller than $A[i]$. Moreover, if we decide to pick some $j < i$ where $A[j] < A[i]$ in the subsequence also, we should pick the longest subsequence that ends in $A[j]$ so that we can also maximize the length of the longest subsequence that ends in $A[i]$. So *if* we decide to go with $A[j]$, then we would get an increasing subsequence of length $LIS(j)+1$ ($+1$ accounts for appending $A[i]$ to it also). As we are interested in taking the longest sequence, we are simply taking a maximum of all available choices.

This concludes the proof of correctness of the formula.

Remark. Let us show why this formula would be wrong *if* we defined $LIS(i)$ to be the length of longest increasing subsequence of the array $A[1 : i]$ *without* the restriction that the subsequence should definitely end in $A[i]$. Consider the following array $A = [1, 2, 3, 7, 2, 5]$:

– $LIS(6) \geq 1 + LIS(5)$ because $A[6] = 5 > 2 = A[5]$ and thus in the line when we are taking maximum, $LIS(5) + 1$ is a valid choice.

– $LIS(5) = 4$ because there is a subsequence $[1, 2, 3, 7]$ of length 4 that belongs to the array $A[1 : 5]$.

– This, combined with the above, implies that $LIS(6) \geq 5$ which is clearly false: the length of the longest subsequence in the array A is at most 4.

As such, this simpler definition of $LIS(i)$ is not good enough to allow for this recursion formula to work. This may act as a reminder of why we need to prove the correctness of the algorithms: on the surface, the two definitions may sound very similar and thus it is more tempting to go with the simpler solution; but we now know that would result in a wrong algorithm.

So we are done with the main step of the solution: we designed a recursive formula for the LIS problem and proved its correctness. In the next lecture, we analyze the runtime of this algorithm (in fact, we will also go over the proof of correctness in more details again).