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

Lecture 11

February 22, 2022

Instructor: Sepehr Assadi

1 Dynamic Programming II: Longest Increasing Subsequence

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 1. 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 $1 \leq i \leq n$, 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).

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:

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

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 $LIS(i)$. Before that, it might be helpful to see an example of how this recursive formula works.

Example: Consider using the formula above to solve the problem on an array $A = [1, 3, 7, 2, 4, 5, 0]$ ($n = 7$):

- $LIS(7)$: computes the length of the LIS ending in $A[7] = 0$:

$$LIS(7) = 1,$$

by definition of the formula since $A[7]$ is smaller than all the other entries.

- $LIS(6)$: computes the length of the LIS ending in $A[6] = 5$:

$$LIS(6) = \max\{1 + LIS(5), 1 + LIS(4), 1 + LIS(2), 1 + LIS(1)\},$$

by definition of the formula since $A[6]$ is larger than $A[5], A[4], A[2], A[1]$, but not $A[3]$.

- $LIS(5)$: computes the length of the LIS ending in $A[5] = 4$:

$$LIS(5) = \max\{1 + LIS(4), 1 + LIS(2), 1 + LIS(1)\},$$

by definition of the formula since $A[5]$ is larger than $A[4], A[2], A[1]$, but not $A[3]$.

- $LIS(4)$: computes the length of the LIS ending in $A[4] = 2$:

$$LIS(4) = \max\{1 + LIS(1)\},$$

by definition of the formula since $A[4]$ is larger than $A[1]$, but not $A[2], A[3]$.

- $LIS(3)$: computes the length of the LIS ending in $A[3] = 7$:

$$LIS(3) = \max\{1 + LIS(2), 1 + LIS(1)\},$$

by definition of the formula since $A[3]$ is larger than $A[2], A[1]$.

- $LIS(2)$: computes the length of the LIS ending in $A[3] = 3$:

$$LIS(2) = 1 + LIS(1),$$

by definition of the formula since $A[2]$ is larger than $A[1]$.

- $LIS(1)$: computes the length of the LIS ending in $A[1] = 1$:

$$LIS(1) = 1,$$

by definition of the formula since $A[1]$ is not larger that any (zero) entries before it.

[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 $A[n+1]$.

We now need to go back and evaluate each of these subproblems:

- $LIS(1) = 1$ (the base case).
- $LIS(2) = 1 + LIS(1) = 2$.
- $LIS(3) = \max\{1 + LIS(2), 1 + LIS(1)\} = 1 + LIS(2) = 3$.
- $LIS(4) = 1 + LIS(1) = 2$.
- $LIS(5) = \max\{1 + LIS(4), 1 + LIS(2), 1 + LIS(1)\} = 1 + LIS(4) = 3$
  (note that we could have also updated $LIS(5)$ from $LIS(2)$ instead of $LIS(4)$).
- $LIS(6) = \max\{1 + LIS(5), 1 + LIS(4), 1 + LIS(2), 1 + LIS(1)\} = 1 + LIS(5) = 4$.
- $LIS(7) = 1$.

The formula then returns $\max_j \{LIS(j) \mid j \in \{1, \ldots, 7\}\} = LIS(6) = 4$ which is the correct answer. Let us also see what is the corresponding subsequence for each value of the formula:

- $LIS(1) = 1$: $A[1] = [1]$.
- $LIS(2) = 1 + LIS(1)$: $A[1], A[2] = [1, 3]$.
- $LIS(3) = 1 + LIS(2)$: $A[1], A[2], A[3] = [1, 3, 7]$.
- $LIS(4) = 1 + LIS(1)$: $A[1], A[4] = [1, 2]$.
- $LIS(5) = 1 + LIS(4)$: $A[1], A[4], A[5] = [1, 2, 4]$.
- $LIS(6) = 1 + LIS(5)$: $A[1], A[4], A[5], A[6] = [1, 2, 4, 5]$.
- $LIS(7) = 1$: $A[7] = [0]$.

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

- If $i = 1$: there is only one subsequence of $A[1 : 1]$ which is $A[1]$ and is increasing on its own, so $LIS(1) = 1$ 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.

Algorithm: We now turn this recursive formula into a memoization and a bottom-up dynamic programming solution and analyze their runtime.

Memoization: We pick an array $T[1:n]$ of length n initialized to be all 'undefined'. We then have the following memoization algorithm.

```
MemLIS($i$):
  1. If $T[i] \neq$ 'undefined' return $T[i]$.
  2. If $i = 1$ let $T[i] = 1$.
  3. Else, let $T[i] = 1$ and for $j = 1$ to $i - 1$:
     (a) If $A[j] < A[i]$, let $T[i] = \max\{T[i], 1 + \texttt{MemLIS}(j)\}$.
  4. Return $T[i]$.
```

The answer to our problem is the obtained by computing $\texttt{MemLIS}(1), \texttt{MemLIS}(2), \ldots, \texttt{MemLIS}(n)$ and then returning the maximum value – in other words, using the following algorithm:

```
  1. Let $m \leftarrow 0$. For $j = 1$ to $n$:
         Let $m \leftarrow \max\{m, \texttt{MemLIS}(j)\}$.
  2. Return $m$.
```

Again, the correctness of this algorithm follows directly from that of the recursive formula $LIS(\cdot)$. As for the runtime, there are $n+1$ subproblems and subproblem $\texttt{MemLIS}(i)$ takes $O(i)$ time (for doing a for-loop). Note that unlike the previous dynamic programming algorithms we saw, now the runtime of each subproblem is different. However, the principle is exactly as before (similar to the case of recurrences): we simply need to add up all the time spent for all subproblems. As such, the total time spent by the algorithm is (we replace $O(i)$ by $C \cdot i$ for some constant C):

$$\sum_{i=0}^{n} C \cdot i = C \cdot \sum_{i=1}^{n} i = C \cdot \frac{n \cdot (n+1)}{2} = O(n^2).$$

So, the runtime of this algorithm is $O(n^2)$ (there is an additional $O(n)$ time at the end to compute the maximum value but that is negligible compared to the $O(n^2)$ term).

Bottom-up dynamic programming: We have to first determine the evaluation order. It is quite easy here because each $LIS(i)$ is updated only from $LIS(i')$ for $i' < i$ and so we simply need to evaluate the subproblems in the increasing order of i.

```
DynamicLIS($A[1:n]$):
  1. Let $T[1:n]$ be an array initialized by 1 originally (to take care of base case).
  2. For $i = 1$ to $n$:
         For $j = 1$ to $i - 1$:
             If $A[j] < A[i]$, let $T[i] = \max\{T[i], T[j] + 1\}$.
  3. Let $m \leftarrow 0$. For $j = 1$ to $n$: Let $m \leftarrow \max\{m, T[j]\}$.
  4. Return $m$.
```

Again the correctness follows from the correctness of the recursive formula $LIS(\cdot)$ and the in the evaluation order we picked, each entry is updated from the cells which are already computed correctly. It is easy to see that the runtime of this algorithm is $O(n^2)$.