CS31 (Algorithms), Spring 2020 : Lecture 8 Supp

Date:

Topic: Dynamic Programming 3: LIS

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

1 Longest Increasing Subsequence

We look at another famous string problem: the LIS, or the Longest Increasing Subsequence problem. In this problem, the input is an array $A[1:n]$ of integers. A subsequence $(A[i_1], A[i_2], \ldots, A[i_k])$ is increasing (actually non-decreasing would be more apt) if $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_k]$. As before, we have $i_1 < i_2 < \cdots < i_k$. Our goal is to find the *longest* increasing subsequence.

For example, suppose the array $A = [13, 27, 15, 8, 19, 32, 20]$. Then the subsequence $(13, 32)$ is an increasing subsequence, so is $(15, 20)$. The longest one seems to be $(13, 15, 19, 20)$. There could be many such LISs of length 4. For example, $(13, 15, 19, 32)$ is another one.

A Wrong Turn. As we have done before, let us consider a solution $\sigma = (A[i_1], A[i_2], \ldots, A[i_k])$ which is an LIS of $A[1:n]$. Once again, if $i_k \neq n$, then σ should be an LIS of $A[1:n-1]$. What can we say if $i_k = n$? We would like to say $\sigma' := \sigma - A[i_n] = (A[i_1], \ldots, A[i_{k-1}])$ is an LIS of a smaller instance. Which smaller subinstance can this be?

An initial gut reaction may to be say that σ' is in fact an optimal solution to $A[1:n-1]$; *but this would be wrong!*. Here is a simple example:

$$A = [10, 30, 20]; \quad \sigma = (10, 20); \quad \sigma' = (10) \text{ which is not an LIS of } [10, 30]$$

What did we miss? We missed that ***an arbitrary*** LIS of $A[1:n-1]$ ***cannot*** be extended by adding $A[n]$ at the end; it only can be extended if the LIS of $A[1:n-1]$ ends with a number at most $A[n]$.

Remark: *Many (including me) have made this mistake above. And indeed, if we argue only in English, it is possible we can fool ourselves. However, if we write the correct proofs, we* will *find the error.*

How do we get out of this impasse? The (clever) idea is to ask for something else from the solution. Given $A[1:n]$ suppose we ask not for an LIS in $A[1:n]$, but for the longest increasing subsequence that *ends* with $A[n]$.

1. Why is this interesting? Because, if we knew this for all $A[1:m]$, for $1 \leq m \leq n$, then we could scan all these n answers and choose the best one.

2. Why is this useful? Now consider the optimal solution for $A[1:n]$; let it be $\sigma = (A[i_1], \ldots, A[i_{k-1}], A[n])$. Now we can claim $\sigma' = (A[i_1], \ldots, A[i_{k-1}])$ is the optimal solution for $A[1:i_{k-1}]$. This is because we are *guaranteed* by σ that $A[i_{k-1}] \leq A[n]$. So any subsequence of $A[1:i_{k-1}]$ which ends with $A[i_{k-1}]$ can be extended by adding $A[n]$. Therefore, we can find the length of σ if we were given all the solutions for the arrays of the form $A[1:j]$ where $A[j] \leq A[n]$. It also makes recovery near trivial.

We are now armed to write the full DP solution. With this new twist of LIS, we are ready for our dynamic programming solution, which we provide below.

1. ***Definition:*** For any $1 \le m \le n$, define $\mathsf{LIS}(m)$ to be the length of the longest increasing subsequence in $A[1:m]$ which ends with $A[m]$. We are interested in $\max_{1 \le m \le n} \mathsf{LIS}(m)$.

2. ***Base Cases:*** $\mathsf{LIS}(0) = 0$.

3. ***Recursive Formulation:*** For all $m \ge 1$:

$$\mathsf{LIS}(m) = 1 + \max_{1 \le j < m : A[j] \le A[m]} \mathsf{LIS}(j)$$

4. ***Formal Proof:*** To formally prove the above, it helps to introduce, as we have done for optimization problems, the notation of $\mathsf{Cand}(m)$ to be the set of all *increasing subsequences* of $A[1:m]$ which end with $A[m]$. Therefore,

$$\mathsf{LIS}(m) = \max_{\sigma \in \mathsf{Cand}(m)} |\sigma|$$

   ($\ge$): Fix any $1 \le j < m$ with $A[j] \le A[m]$, and let σ' be the LIS of $A[1:j]$ ending with $A[j]$ of length $\mathsf{LIS}(j)$. Well $\sigma = \sigma' \circ A[m]$ is in $\mathsf{Cand}(m)$ and is of length $1 + \mathsf{LIS}(j)$. This means, $\mathsf{LIS}(m) \ge 1 + \mathsf{LIS}(j)$.

   ($\le$): Fix the sequence $\sigma \in \mathsf{Cand}(m)$ of length $\mathsf{LIS}(m)$. If $|\sigma| = 1$, then the inequality is vacuous. Otherwise, let $A[j]$ be the second-last entry of σ. Note that $\sigma' = \sigma - A[m]$ is in $\mathsf{Cand}(j)$ and has length $\mathsf{LIS}(m) - 1$.

5. ***Pseudocode for computing*** $\mathsf{LIS}(m)$ ***and recovery pseudocode:***

```
1:  procedure LIS(A[1:n]):
2:      ▷ Returns the Longest Increasing Subsequence
3:      Allocate space L[0:n] ▷ L[m] will contain LIS(m).
4:      Maintain parent[1:n] which is useful for recovery. Initialized to 0
5:      L[0] = 0. ▷ Base Case.
6:      for 1 ≤ m ≤ n do:
7:          L[m] = 1 + max_{1≤j<m:A[j]≤A[m]} L[j] ▷ Naively, takes O(n) time.
8:          Set parent[m] to be the j which maximizes. If no such j, parent remains 0.
9:      m = arg max L[1:n] ▷ L[m] contains the length of the optimal LIS
10:     Define σ to be the reverse of (A[m], A[parent[m]], A[parent[parent[m]]], ..., ) till
        parent becomes 0.
11:     return σ.
```

6. ***Running time and space*** As written above, the space is $O(n)$, and the running time is $O(n^2)$ since Line 7 takes $O(n)$ time when implemented naively. Indeed, the naive implementation scans the whole array $A[1:m-1]$ and for each j such that $A[j] \le A[m]$, it stores the $L[j]$, and then finds the largest among these.

   Is this the best implementation? The answer is No! And before reading on, you should think for sometime on how you would solve it faster. One can in fact solve this in $O(\log n)$ time. However, one needs to keep maintaining added information as we go along. The $O(\log n)$ should smell of binary search. But L is not an increasing array ... Take a peek at the next page only after thinking a while.

Improved Running Time. We maintain two data structures as the above algorithm progresses. One is a *sorted strictly increasing* array[1] T contains all the different LIS values observed. That is, T is the sorted order of the unique elements of the array $L[]$. Initially it contains only the entry 0.

The second is a dictionary D which maps every element $t \in T$ to an element $A[j]$ in the array. There are two important properties:

(a) $\mathsf{LIS}(j) = t$, that is, j is one of the locations where LIS takes the value t, and

(b) among all the other (if any) i's with $\mathsf{LIS}(i) = t$, $A[j] \leq A[i]$.

Here is the crucial observation.

Claim 1. If $t < t'$, then $D(t) \leq D(t')$.

Proof. Suppose not. Suppose $D(t) = A[j]$ and $D(t') = A[j']$ and $A[j'] < A[j]$. We will reach a contradiction. Note, that $D(t') = A[j']$ implies $\mathsf{LIS}(j') = t'$. That is, there is a sequence $(i_1, i_2, \ldots, i_{t'} = j')$ such that $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_{t'}]$. Now, since $t < t'$, if we note that there is a subsequence $A[i_1] \leq A[i_2] \leq \cdots \leq A[i_t]$ inside this. Note that $\mathsf{LIS}(i_t) = t$ as well. However, $A[i_t] \leq A[j'] < A[j]$. This contradicts the fact that $D(t) = A[j]$, since $D(t)$ is supposed to point to the smallest $A[j]$ with $\mathsf{LIS}(j) = t$. □

Before we go on to show how (T, D) are maintained as the algorithm progresses, let us see how (T, D) help in speeding up Line 7. Recall, given m, we need to find

$$\max_{1 \leq j < m : A[j] \leq A[m]} \mathsf{LIS}(j)$$

Suppose this maximum value if t^*. Note $t^* \in T$. Let j^* be such that $D(t^*) = A[j^*]$. By definition $A[j^*] \leq A[m]$. The algorithm to find j^* would binary search over T. Note that the previous claim implies that if $D(t) > A[m]$, then $t^* < t$. If not, then $D(t^*) \geq D(t) > A[m]$ which contradicts $D(t^*) \leq A[m]$. Therefore, by binary search we can find the *largest* t^* such that $D(t^*) \leq A[m]$, and that would be our answer.

Lastly, we need to show how (T, D) are maintained. This is easier. As soon as $\mathsf{LIS}(m)$ is calculated for some m, we check if $t = \mathsf{LIS}(m)$ is in T. If not, we add t to T in the sorted order (this is where we need the binary-search tree implementation) in $O(\log n)$ time, and set $D(t) = A[m]$. Otherwise, we check if $A[m] < D(t)$, and in that case set $D(t) = A[m]$. Otherwise, do nothing. That completes the description of the $O(\log n)$ implementation of Line 7. Which in turn leads to the following theorem.

Theorem 1. The Longest Increasing Subsequence can be solved in $O(n \log n)$ time.

[1] Techincally, this is a binary search tree which allows insertion and search