# COMP 182: Algorithmic Thinking
# Dynamic Programming

Luay Nakhleh

## 1 Formulating and understanding the problem

The LONGEST INCREASING SUBSEQUENCE, or LIS, Problem is defined as follows.

- **Input:** An array $A[0..n-1]$ of integers.
- **Output:** A sequence of indices $(i_1, i_2, \ldots, i_k)$ that satisfies the following conditions:
  - *Feasibility condition 1*: $i_1 \geq 0$, $i_k \leq n-1$, and $\forall 1 \leq j < k,\ i_j < i_{j+1}$.
  - *Feasibility condition 2*: $\forall 1 \leq j < k, A[j] < A[j+1]$.
  - *Optimality condition*: For every sequence of indices $(i_1, i_2, \ldots, i_\ell)$ that satisfies feasibility conditions 1 and 2, $k \geq \ell$.

Let us illustrate the problem on input array $A = [5, 2, 8, 6, 3, 6, 9, 7]$. In this case, $n = 8$. Let us first illustrate feasible solution candidates that are not necessarily optimal. This amounts to finding sequences of indices $(i_1, i_2, \ldots, i_k)$ that satisfy the two feasibility conditions, but not necessarily the optimality condition. All of the following sequences of indices do satisfy the two feasibility conditions:

1. $()$.
2. $(0)$.
3. $(4)$.
4. $(0, 7)$.
5. $(0, 5, 6)$.
6. $(0, 5, 7)$.
7. $(1, 4, 5, 6)$.
8. $(1, 4, 5, 7)$.

Of course, there are many other sequences of indices that satisfy the two feasibility conditions. Of all the sequences that do satisfy the two feasibility conditions, it turns out that the sequences numbered 7 and 8 in the list above both satisfy that optimality condition in the problem definition. Therefore, one possible solution to the LIS problem on the given input in $(1, 4, 5, 6)$, and another possible solution is $(1, 4, 5, 7)$. That is, while the length of the LIS is unique, the LIS itself might not be unique. This is why to be completely accurate in defining the problem above, we wrote in describing the output "*A* sequence of indices...", rather than "*The* sequence of indices...".

To test whether you understand the problem definition, think about the following questions.

**Question 1** *Consider an array $A[0..n-1]$ of distinct integers.*

1. *Assume $A$ is sorted in increasing order. How many sequences of indices that satisfy the two feasibility conditions exist? How many sequences of indices that satisfy the feasibility and optimality conditions exist? What is the length of an LIS of $A$?*
2. *Assume $A$ is sorted in decreasing order. How many sequences of indices that satisfy the two feasibility conditions exist? How many sequences of indices that satisfy the feasibility and optimality conditions exist? What is the length of an LIS of $A$?*

## 2 Reasoning about a solution

Recursively reasoning about solving this problem centers around one fundamental question:

For an arbitrary index $i$, is $A[i]$ part of an LIS of $A$?

Imagine that we have an oracle $M$ that can answer the question. That is, $M$ on pair $(i, A)$ returns YES whenever $A[i]$ is part of an LIS of $A$ and returns NO whenever $A[i]$ is not part of an LIS of $A$. Using this oracle, one can think of solving the problem as follows (where $lis$ is the solution):

1. $lis \leftarrow ()$;
2. **for** $i \leftarrow 0$ **to** $n - 1$

    **if** $M(i, A) =$YES **then**

        Append $i$ to the right end of $lis$;

However, there is a problem with this algorithm: It is incorrect. The only condition under which this algorithm would work is if the LIS of input array $A$ was unique. For example, consider the array $A = [5, 2, 8, 6, 3, 6, 9, 7]$ that we used in Section 1 above. For this example, the oracle $M$ would return YES for the following values of the index $i$: 1, 4, 5, 6, 7. However, the sequence $(1, 4, 5, 6, 7)$ of indices does not form an LIS of $A$ since it violates feasibility condition 2 in the problem definition.

So, what caused this problem? In other words, if $A[i]$ is part of an LIS of $A$, how could it be that the list of all such $i$'s is not a solution. This problem stems from the fact that an LIS of an array $A$ is not necessarily unique. So, when more than one LIS exists, the above algorithm would simply merge all solutions and return a "super list" of the indices of all the LIS's, and this super list is not an LIS. To remedy this problem, the algorithm must have knowledge of an LIS that is being constructed and not just add elements to it regardless of that LIS. This gives rise to the following correct algorithm.

1. $lis \leftarrow ()$;
2. **for** $i \leftarrow 0$ **to** $n - 1$

    **if** $M(i, A) =$YES **and** $A[i] > A[j]$ for every $j$ in $lis$ **then**

        Append $i$ to the right end of $lis$;

This algorithm would now correctly return $(1, 4, 5, 6)$ on the array $A = [5, 2, 8, 6, 3, 6, 9, 7]$.

Observe that at every point of the execution of this algorithm, if $lis = (i_1, i_2, \ldots, i_m)$, then $A[i_1] < A[i_2] < \cdots < A[i_m]$. In other words, $A[i_m]$ is greater than every other element of $A$ indexes by $i_1, i_2, \ldots, i_{m-1}$. This allows us to modify the above algorithm, without affecting its correctness, as follows.

1. $lis \leftarrow ()$;
2. **for** $i \leftarrow 0$ **to** $n - 1$

    **if** $M(i, A) =$YES **and** $A[i] > A[j]$ where $j$ is the rightmost element in $lis$ **then**

        Append $i$ to the right end of $lis$;

The question now is: how do we obtain an algorithm that doesn't make use of some magic oracle $M$? While we will do without the oracle itself, the reasoning we have employed thus far is key to figuring out the algorithm.

## 3 We don't need an oracle: Recursion to the rescue

Notice that since the fundamental question above referred to an *arbitrary* index $i$, we can set the value $i$ to whatever makes most sense to us. I'd argue that it is most natural to take $i$ to be either 0 or $n - 1$; that is, to look at either the first or last element of the array $A$ and ask whether that element is in the LIS of $A$. So, let us now ask the question in terms of the first element:

Is $A[0]$ part of an LIS of $A$?

We now have one of two scenarios:

- Scenario 1: The answer to the question is NO, in which case we can ignore index 0 and look for a solution on $A[1..n-1]$.
- Scenario 2: The answer to the question is YES, in which case the solution is index 0 in addition to a solution on $A[1..n-1]$.

However, observe that in Scenario 2, since index 0 is part of the solution, when we look for the "remainder" of the solution in $A[1..n-1]$, we must respect the fact that *all* elements of the solution on $A[1..n-1]$ must satisfy feasibility condition 2 in the problem definition with respect to index 0. That is, if $lis$ is the solution on $A[1..n-1]$ and index 0 is part of the solution, then $A[0] < A[j]$ for every $j$ in $lis$. As we reasoned above, this is equivalent to stating that $A[0] < A[j]$ where $j$ is the first (leftmost) element of $lis$. Therefore, we revise the two scenarios as follows:

- Scenario 1: The answer to the question is NO, in which case we can ignore index 0 and look for a solution on $A[1..n-1]$.
- Scenario 2: The answer to the question is YES, in which case the solution is index 0 in addition to a solution on $A[1..n-1]$ where the first (leftmost) element $j$ of the solution on $A[1..n-1]$ satisfies $A[0] < A[j]$.

Let us write this in a more mathematical formalism. Denote by $LIS(A)$ a LIS of array $A$. For a list $L$, we denote by $L[0]$ the first element of the list, and for two lists $L_1$ and $L_2$, we denote by $L_1 + L_2$ appending list $L_2$ to the right end of $L_1$. For example, $[1,3,6] + [4,5] = [1,3,6,4,5]$. Using this notation, we can write the above two scenarios for finding $LIS(A)$ as:

- Scenario 1: If $A[0]$ is not part of a $LIS$ of $A[0..n-1]$, then we just compute $LIS(A[1..n-1])$.
- Scenario 2: If $A[0]$ is part of of a $LIS$ of $A[0..n-1]$, then $LIS(A[0..n-1]) = [0] + LIS'(A[0], A[1..n-1])$.

Notice that we have introduced $LIS'(A[0], A[1..n-1])$ for Scenario 2, since in this case we want the solution on $A[1..n-1]$ to respect the fact that index 0 is part of the solution. To avoid having two version of the function, $LIS$ and $LIS'$, we notice that $LIS(A) = LIS'(-\infty, A)$. We now just make use of $LIS'$, but remove the prime and call it $LIS$, which gives us the following two scenarios:

- Scenario 1: If $A[0]$ is not part of a $LIS$ of $A[0..n-1]$, then we just compute $LIS(-\infty, A[1..n-1])$.
- Scenario 2: If $A[0]$ is part of of a $LIS$ of $A[0..n-1]$, then $LIS(A[0..n-1]) = [0] + LIS(A[0], A[1..n-1])$.

Which of these two scenarios results in an optimal solution to the problem? Well, it is easy to answer this question: Evaluate both of them, and take the one that results in a longer subsequence! In other words, if we use $|L|$ to denote the length of list $L$, we have

$$|LIS(A[0..n-1])| = \max\{|LIS(A[1..n-1])|, |[0] + LIS(A[1..n-1])|\}. \quad (1)$$

The blue term corresponds to Scenario 1, and the red term corresponds to 2.

# 4 A straightforward recursive implementation and its prohibitive running time

We now make use of the recursive formula (1) to devise a recursive algorithm for computing the LIS of an array $A$. On the slides, we saw a version of the algorithm that returns the length of the LIS; here, we describe it in terms of returning the LIS itself. The pseudo-code of the algorithm is given in Algorithm 1.

We illustrate the recursive algorithm on input array $[3,6,9,7]$ in Fig. 1.

What is the running time of Algorithm *Rec-LIS*? Clearly, it equals the running time of Algorithm *LIS-helper*! So, for an array $A$ of size $n$, what is the running time $T(n)$ of Algorithm **LIS-helper**?

Except for Lines 1 and 2 in Algorithm **LIS-helper**, the algorithm can be implemented so that all other lines take a constant number of operations, $c$. Each of the two **LIS-helper** calls on Lines 1 and 2 takes $T(n-1)$ time. Therefore, a recurrence for the running time of this algorithm is

$$T(n) = 2T(n-1) + c.$$

**Algorithm 1: Rec-LIS**

**Input**: Array $A[0..n-1]$ of integers.
**Output**: A LIS of $A[0...n-1]$.

**return LIS-helper**$(-\infty, A[0..n-1], 0)$;

**Algorithm 2: LIS-helper**

**Input**: Integer $k$, array $A$ of integers, and integer $i$.
**Output**: A LIS of $A[i...|A|-1]$ where every element in the LIS is larger than $k$.

**if** $i \geq |A|$ **then**
  **return** [];
**else**
1  $L \leftarrow$**LIS-helper**$(k, A, i+1)$; // Scenario 1
  **if** $A[i] > k$ **then**
2  $L' \leftarrow [i]+$**LIS-helper**$(A[i], A, i+1)$; // Scenario 2
    **if** $|L'| > |L|$ **then**
      $L \leftarrow L'$;
**return** $L$;

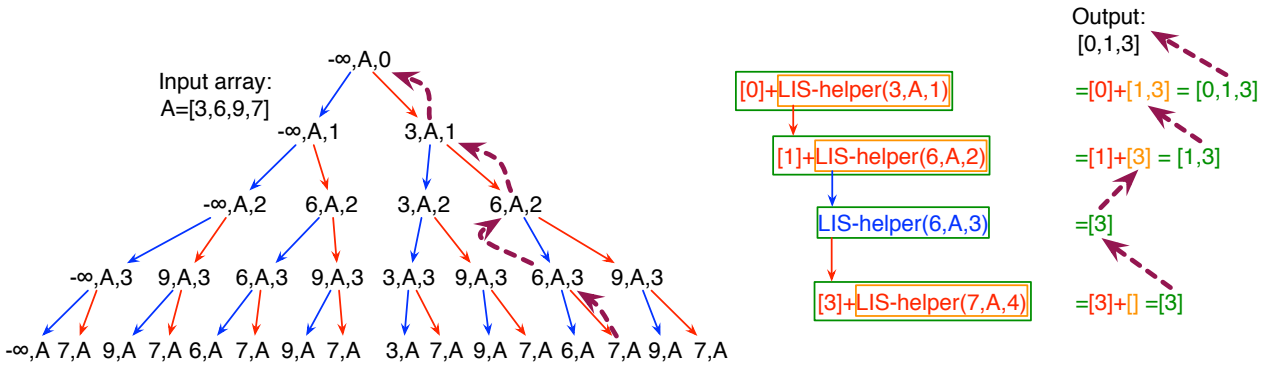


Figure 1: The recursion tree of Algorithm **Rec-LIS** when called on input $(-\infty, [3,6,9,7], 0)$. Shown in every node of the tree is the pair of parameters $k, A$ that is passed to the **LIS-helper** function call. The blue arrows correspond to Scenario 1 (the blue function call in Algorithm **LIS-helper**) and the red arrows correspond to Scenario 2 (the red function call in Algorithm **LIS-helper**). In the last layer of the tree (the leaves), the third parameter passed to the function is not shown; it is 4 for all nodes. Shown on the right is the information on the path in the tree that results in the optimal solution (obtained by going backward from the leaf with parameters $(7, A, 4)$ to the root, following the dashed maroon arrows).

To find a closed formula for this recurrence, we can use an iterative approach as follows:

$$\begin{aligned} T(n) &= 2T(n-1)+c \\ &= 2(2T(n-2)+c)+c \\ &= 2(2(2T(n-3)+c)+c)+c \\ &= \dots \\ &= O(2^n) \end{aligned}$$

Therefore, this algorithm is exponential in the size of the input. This should make sense in light of the illustration in Fig. 1. The recursion tree on the array $A$, whose size is 4, is a complete binary tree whose height is 4. The number of nodes in the tree is $2^5 - 1 = 31$. For an array of size $n$, the tree has $2^{n+1} - 1$ nodes, which explains the exponential running time we obtain.

# 5 Intelligent implementation of the recursion: A dynamic programming algorithm

The recursion tree in Fig. 1 exhibit three features of the solution that allow us to think of a more efficient implementation:

1. To obtain a solution on input $(k, A, i)$, the algorithm makes use of solutions on inputs $(k, A, i+1)$ and $(A[i], A, i+1)$. In other words, the solution to the problem can be obtained from solutions to sub-problems. This is typical of all recursive algorithms.

2. There is a natural order in which sub-problems must be solved: First solve for the inputs in the tree leaves, then solve for the inputs one layer above the leaves, etc., until the problem is solved for the input in the root. This, too, is typical of recursive algorithms.

3. There are many overlapping sub-problems. For example, the sub-problem given by the input $(9, A, 3)$ is shared by the problems given by the inputs $(-\infty, A, 2)$, $(6, A, 2)$ and $(3, A, 2)$. This is *not* typical of all recursive algorithms. For example, in Algorithm **MergeSort** that we had seen, every recursive call of the algorithm is invoked on an instance of the input that is distinct from any other instance that the algorithm is invoked on.

These three features point to the following directions to improve the implementation:

1. Solve the problems from the leaves of the tree towards the root, and

2. Whenever the problem is solved on an input $(k, A, i+1)$, store it, and make use of it, rather than recompute it, whenever it is needed.

Notice that when calling the function on input $(k, A, j)$, $k$ is always an element $A[i]$ for some $0 \leq i \leq n-1$. The only exception is the case where $k = -\infty$. For convenience, and to avoid making this distinction between $-\infty$ and elements in $A$, we can assume that $A$ is of size $n+1$, where $A[0] = -\infty$ and $A[1..n]$ contain the $n$ (finite) integers. We can now introduce a matrix $LISM$ of dimensions $(n+1) \times (n+2)$ such that $LISM[i, j]$ is the length of the LIS of $A[j..n]$ such that all the elements of such a LIS are greater than $A[i]$.

Looking again at the tree in Fig. 1, we can see that if a node is labeled with $(A[i], A, j)$, its two child nodes are labeled with $(A[i], A, j+1)$ and $(A[j], A, j+1)$. If we use the values stored in the matrix $LISM$, this observation tells us that $LISM[i, j]$ is a function of two values: $LISM[i, j+1]$ and $LISM[j, j+1]$.

Another observation is that the $(A[j], A, j+1)$ child node corresponds to a "red function call", which corresponds to a solution of which $A[i]$ is part.

The above two observations, coupled with the fact that every node in the tree takes the maximum of the values computed by its two children, we obtain the general formula:

$$LISM[i, j] = \max\{LISM[i, j+1], 1 + LISM[j, j+1]\}.$$

However, if $A[i] \not< A[j]$, then $A[j]$ cannot be put in a solution that already includes $A[i]$ and the algorithm would have to skip and check if $A[j+1]$ is part of the solution.

Finally, for the boundary condition where $j > n$, it is easy to see that $A[i, j] = 0$. All these results give us a general formula as follows:

$$LISM[i, j] \leftarrow \begin{cases} 0 & \text{if } j > n \\ LISM[i, j+1] & \text{if } A[i] \geq A[j] \\ \max\{LISM[i, j+1], 1 + LISM[j, j+1]\} & \text{otherwise} \end{cases}$$

The full algorithm that computes these values is given in Algorithm 3. The first **for** loop takes care of the boundary condition $j > n$ for all values of $i$. Looping over values of $j$ from $n$ down to 1 reflects the order of evaluating the sub-problems (from the leaves of the recursion tree up to the root). Notice that entries in column 0 of matrix $LISM$ are not filled, since they are not needed. Finally, the algorithm returns $LISM[0, 1]$ which is the length of the LIS of $A[1..n]$ where every element of the LIS is greater than $A[0]$ (which is $-\infty$).

The matrix $LISM$ that Algorithm **DP-ILS** computes on input array $A = [3, 6, 9, 7]$ is:

**Algorithm 3: DP-LIS**

**Input**: Array $A[0..n]$ of integers, where $A[0] = -\infty$.
**Output**: Length of LIS of $A[1...n]$.

```
for i ← 0 to n do
    LISM[i, n + 1] ← 0;
for j ← n downto 1 do
    for i ← 0 to j − 1 do
        if A[i] ≥ A[j] then
            LISM[i, j] ← LISM[i, j + 1];
        else
            LISM[i, j] ← max{LISM[i, j + 1], 1 + LISM[j, j + 1]};
return LISM[0, 1];
```

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | | 3 | 2 | 1 | 1 | 0 |
| 1 | | | 2 | 1 | 1 | 0 |
| 2 | | | | 1 | 1 | 0 |
| 3 | | | | | 0 | 0 |
| 4 | | | | | | 0 |

Notice that $A[0, 1]$ has the value 3, which is the length of an LIS of the given array $A$.

To obtain an LIS of $A$ (not just the length of an LIS), we can use the traceback technique. Algorithm **ComputeLIS** below makes use of $A$ and the matrix $LISM$ that has been computed by Algorithm **DP-LIS** to compute an LIS of $A$.

**Algorithm 4: ComputeLIS**

**Input**: Array $A[0..n]$ of integers, where $A[0] = -\infty$, and matrix $LISM$.
**Output**: An LIS $lis$ of $A[1...n]$.

```
lis ← [];
k ← 0;
i ← 0;
for j ← 1 to n do
    if LISM[i, j] = 1 + LISM[j, j + 1] then
        lis[k] = j;
        i ← j;
        k ← k + 1;
return lis;
```

It takes $O(n^2)$ time to compute an LIS of an array $A$ that has $n$ elements since Algorithm **DP-LIS** takes $O(n^2)$ time and Algorithm **ComputeLIS** takes $O(n)$ time. Contrast this to the exponential running time of the recursive algorithm for the same problem!