1 Designing a DP for Longest Increasing Subsequence (LIS)

Given a sequence of numbers, we want to find a strictly increasing subsequence of it that is also the longest. The numbers in the subsequence may not be consecutive in the original sequence. For example, given sequence \( a = \{4, 2, 5, 3, 9, 7, 8, 10, 6\} \), its LIS is \( \{2, 5, 7, 8, 10\} \) or \( \{2, 3, 7, 8, 10\} \), as they both have length 5.

1.1 A Failed Attempt

A natural subproblem is to have \( f[i] \) denote the length of the LIS of sequence \( a[1...i] \). A natural transition function is to consider whether the LIS of \( a[1...i] \) should include \( a[i] \) or not, and take max of the two.

If \( a[i] \) is not included, then simply \( f[i] = f[i-1] \). If \( a[i] \) is included, however, we run into a problem: when the last element \( a[i] \) is in the sequence, we have the additional constraint that all other elements need to be smaller than \( a[i] \). However, when we reference a previous subproblem \( f[j] \) where \( j < i \), we do not know whether the solution for \( f[j] \) uses numbers strictly smaller than \( a[i] \), hence our proposed transition function does not work.

1.2 Attempt 2

Consider the following subproblem definition: Let \( f[i] \) denote the length of the LIS of sequence \( a[1...i] \) that ends at \( a[i] \). (i.e. the subsequence must include \( a[i] \))

The decision at \( f[i] \) is immediate, as we have to pick \( a[i] \) by definition. To compute \( f[i] \), we can enumerate the number that is before \( a[i] \) in the sequence. This motivates our transition function:

\[
f[i] = \max\{1, \max_{j<i, a[j]<a[i]} f[j] + 1\}
\]

If the max evaluates to the first case then the subsequence is simply \( \{a[i]\} \); if it evaluates to the second case then the subsequence is \( \{\text{LIS ending at } a[j], a[i]\} \).

For example, for the sequence mentioned above, we would fill out a DP table like below
\[ a[i] = \{4, 2, 5, 3, 9, 7, 8, 10, 6\} \]

<table>
<thead>
<tr>
<th>(i)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f[i])</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

To complete our algorithm, we also need a base case that is \(f[0] = 0\), and an output that is \(\max_{1 \leq i \leq n} f[i]\).

### 1.2.1 Analyze Running Time

The running time of a DP, in general, is

\[
\text{# states} \times \text{time for evaluating one transition function}
\]

In the DP above, there are \(n\) states, and we take \(O(n)\) to evaluate one transition function. Hence the total running time is \(O(n^2)\).

### 1.2.2 Proof of Correctness

We will use induction to prove that our DP computes the correct answer. Our inductive hypothesis, in general, is to assume that “smaller subproblems are computed correctly.”

- **Base case:** \(f[0] = 0\) is true by definition.
- **Inductive hypothesis:** assume that for every \(j < i\), \(f[j]\) is indeed the length of the LIS ending at \(a[j]\).
- **Induction step:** Let \(b[i]\) denote the LIS ending at \(a[i]\). \(b[i]\) is either of length 1 or of length greater than 1.
  - If \(b[i]\) is of length 1, then it is considered by the first case of the transition function.
  - If \(b[i]\) is of length greater than 1, let \(a[j]\) denote the second-to-last number in \(b[i]\). By definition \(j < i\) and \(a[j] < a[i]\). By IH, \(f[j]\) is computed correctly. Hence \(f[i] = f[j] + 1\) is considered by the second case of the transition function.

Therefore, \(f[i]\) is also computed correctly.
- **By induction,** \(f[i]\) is computed correctly for all \(i \geq 0\).