COMPSCI 330: Design and Analysis of Algorithms **Jan 24, 2019**

Lecture 5: Dynamic Programming II

Lecturer: Rong Ge *Scribe: Haoming Li*

Overview

In this lecture, we apply the idea of dynamic programming to the problem of Longest Increasing Subsequence (LIS) and Longest Common Subsequence (LCS).

5.1 Longest Increasing Subsequence

In the longest increasing subsequence problem, the input is a sequence of numbers $\{a_1, \ldots, a_n\}$. A subsequence is any subset of these numbers taken in order, of the form $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ where $1 \le i_1 < i_2 < \ldots i_k \le n$, and an increasing subsequence is one in which the numbers are getting strictly larger. The task is to find the length of the increasing subsequence of greatest length. For instance, the longest increasing subsequence of {4, 2, 3, 6, 9, 7} is {2, 3, 6, 7} whose length is 4.

5.1.1 Idea 1: 2D Dynamic Program

If we think of the problem as making a sequence of decisions: for each number, decide whether it is in the LIS. Focusing on the last decision, we can choose to whether put the last number in the LIS or leave it out.

We then try to relate each option to a smaller subproblem. Reusing the example above, if we choose to leave the last number out, then the subproblem becomes finding (the length of) the LIS in sequence {4, 2, 3, 6, 9}; if we choose to put the last number in , then the subproblem becomes finding (the length of) the LIS in sequence {4, 2, 3, 6, 9} while making sure that the numbers used in the rest of the LIS are less than 7.

This idea can be implemented with a 2D-table dynamic program.

5.1.2 Idea 2: 1D Dynamic Program

Following the same idea of focusing on the last decision, we can also arrive at another approach. If we choose to always put the last number in the LIS, then the subproblem becomes finding the longest LIS that ends at the last number. We are now ready to present the state, the transition function, and, last but not least, the algorithm that uses this idea. Let $a[\ ]$ denote the input sequence.

- **State:** Let $b[i]$ denote the length of the LIS that ends at $a[i]$;
- **Transition function:** $b[i] = (\max_{1 \le j < i, a[j] < a[i]} b[j]) + 1$
- **Algorithm:**

```
$b[1] = 1$;
for $i = 2 \ldots n$:
  $b[i]$ = Transition function;
return $\max_{i \le n} b[i]$;
```

Below is, by the end of the algorithm, what the 1×6 dynamic programming table $b[\,]$ look like if the input was $a[\,] = \{4, 2, 3, 6, 9, 7\}$:

1	1	2	3	4	4

This algorithm runs in $O(n^2)$ time, as for-loop takes linear-time and, within the for-loop, max function takes linear-time.

5.2 Longest Common Subsequence

In the longest common subsequence problem, the input are two strings; the definition of subsequence is the same as in LIS (can skip characters). The goal is to find the length of the common subsequence of greatest length. For instance, the longest common subsequence of $a[\,]$ = '*ababcde*' and $b[\,]$ = '*abbecd*' is '*abbcd*', whose length is 5.

5.2.1 A Dynamic Program

Again, think of the problem a making a sequence of decisions: for any pair of characters, decide whether they are match in the LCS. Focus on the last decision and enumerate the options: for the last characters of both sequence, they are either both in LCS, or not.

We then try to relate each option to a smaller subproblem. Reusing the example above, we cannot have $a[n]$ matching $b[m]$ in the LCS, thus they cannot both be in the LCS. If $a[n]$ is not in the LCS, then subproblem becomes finding the LCS of '*ababcd*' and '*abbecd*'; if $b[m]$ is not in the LCS, then subproblem becomes finding the LCS of '*ababcde*' and '*abbec*'. With this ides in mind, we are now ready to present the dynamic program.

- **State:** Let $m[i, j]$ denote the length of the LCS of $a[1 \dots i]$ and $b[1 \dots j]$
- **Transition function:** $m[i, j] = \max \begin{cases} m[i-1, j-1] + 1, & \text{if } a[i] = b[j] \text{ (both in LCS)} \\ m[i-1, j], & \text{if } a[i] \neq b[j] \text{ (} a[i] \text{ not in LCS)} \\ m[i, j-1], & \text{if } a[i] \neq b[j] \text{ (} b[j] \text{ not in LCS)} \end{cases}$
- **Algorithm:**

```
$m[0, j] = 0$; $m[i, 0] = 0$;
for $i = 1 \dots n$:
  for $j = 1 \dots m$:
    $m[i, j]$ = Transition function;
return $m[n, m]$;
```

This algorithm runs in $O(nm)$ time, as double-nested for-loop takes quadratic time and within the for-loop, max function takes constant time. We now prove the correctness of the algorithm by induction. The idea is to do induction in the same order as you compute the states.

- **Proof of correctness:**
  - Induction Hypothesis: the algorithm computes $m[i, j]$ correctly for all $(i, j) < (u, v)$. ($m[i, j]$ is computed before $m[u, v]$.)
  - Base case: $m[0, j] = m[i, 0] = 0$. Correct because empty sequence has no LCS.
  - Inductive Step: Assume IH is true. When computing $m[u, v]$, the transition considers 3 cases. By IH, $m[u-1, v-1]$, $m[u-1, v]$, $m[u, v-1]$ are computed correctly. Hence, the algorithm makes the correct decision and $m[u, v]$ is computed correctly. QED