CS 6901 (Applied Algorithms) – Lecture 11

Antonina Kolokolova*

October 25, 2016

0.1 Longest Common Subsequence

The input consists of two sequences $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_m$. The goal is to find a longest common subsequence of $\vec{x}$ and $\vec{y}$, that is a sequence $z_1, \ldots, z_k$ that is a subsequence both of $\vec{x}$ and of $\vec{y}$. Note that a sub*sequence* is not always sub*string*: if $\vec{z}$ is a subsequence of $\vec{x}$, and $z_i = x_j$ and $z_{i+1} = x_{j'}$, then the only requirement is that $j' > j$, whereas for a sub*string* it would have to be $j' = j + 1$.

For example, let $\vec{x}$ and $\vec{y}$ be two DNA strings $\vec{x} = TGACTA$ and $\vec{y} = GTGCATG$; $n = 6$ and $m = 7$. Then one common subsequence would be GTA. However, it is not the longest possible common subsequence: there are common subsequences $TGCA$, $TGAT$ and $TGCT$ of length 4.

To solve the problem, we notice that if $x_1 \ldots x_i$ and $y_1 \ldots y_j$ are prefixes of $\vec{x}$ and $\vec{y}$ respectively, and $x_i = y_j$, then the length of the longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$ is one plus the length of the longest common subsequence of $x_1 \ldots x_{i-1}$ and $y_1 \cdots y_{j-1}$.

Step 1. We define an array to hold partial solution to the problem. For $0 \leq i \leq n$ and $0 \leq j \leq m$, $A(i, j)$ is the length of the longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$. After the array is computed, $A(n, m)$ will hold the length of the longest common subsequence of $\vec{x}$ and $\vec{y}$.

Step 2. At this step we initialize the array and give the recurrence to compute it.

*This set of notes is based on the course notes of U. of Toronto CS 364 as taught by Stephen Cook

For the initialization part, we say that if one of the two (prefixes of) sequences is empty, then the length of the longest common subsequence is 0. That is, for $0 \leq i \leq n$ and $0 \leq j \leq m$, $A(i, 0) = A(0, j) = 0$.
The recurrence has two cases. The first is when the last element in both subsequences is the same; then we count that element as part of the subsequence. The second case is when they are different; then we pick the largest common sequence so far, which would not have either x_i or y_j in it. So, for $1 \leq i \leq n$ and $1 \leq j \leq m$,

$A(i, j)$ for the above example.

	∅	G	T	G	C	A	T	G
∅	0	0	0	0	0	0	0	0
T	0	0	1	1	1	1	1	1
G	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
C	0	1	1	2	3	3	3	3
T	0	1	2	2	3	3	4	4
A	0	1	2	2	3	4	4	4

$$A(i,j) = \begin{cases} A(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max\{A(i-1, j), A(i, j-1)\} & \text{if } x_i \neq y_j \end{cases}$$

Step 3. Skipped.

Step 4. As before, just retrace the decisions.

0.2 Longest Increasing Subsequence

Now let us consider a simpler version of the LCS problem. This time, our input is only one sequence of distinct integers $\vec{a} = a_1, a_2, \ldots, a_n.$, and we want to find the longest *increasing* subsequence in it. For example, if $\vec{a} = 7, 3, 8, 4, 2, 6$, the longest increasing subsequence of $\vec{a}$ is $3, 4, 6$.

The easiest approach is to sort elements of $\vec{a}$ in increasing order, and apply the LCS algorithm to the original and sorted sequences. However, if you look at the resulting array you would notice that many values are the same, and the array looks very repetitive. This suggest that the LIS (longest increasing subsequence) problem can be done with dynamic programming algorithm using only one-dimensional array.

Step 1: Describe an array of values we want to compute.
For $1 \leq i \leq n$, let $A(i)$ be the length of a longest increasing sequence of $\vec{a}$ that end with a_i. Note that the length we are ultimately interested in is $\max\{A(i) \mid 1 \leq i \leq n\}$.

Step 2: Give a recurrence.
For $1 \le i \le n$,
$A(i) = 1 + \max\{A(j) \mid 1 \le j < i \text{ and } a_j < a_i\}$.
(We assume $\max \emptyset = 0$.)
We leave it as an exercise to explain why, or to prove that, this recurrence is true.
Step 3: Give a high-level program to compute the values of A.
This is left as an exercise. It is not hard to design this program so that it runs in time $O(n^2)$. (In fact, using a more fancy data structure, it is possible to do this in time $O(n \log n)$.)

LCS and LIS arrays for the example

A(i,j)	$\emptyset$	7	3	8	4	2	6
$\emptyset$	0	0	0	0	0	0	0
2	0	0	0	0	0	**1**	1
3	0	0	**1**	1	1	1	1
4	0	0	1	1	**2**	2	2
6	0	0	1	1	2	2	**3**
7	0	**1**	1	1	2	2	3
8	0	1	1	**2**	2	2	3
$A(i)$		**1**	**1**	**2**	**2**	**1**	**3**

Step 4: Compute an optimal solution.
The following program uses A to compute an optimal solution. The first part computes a value m such that $A(m)$ is the length of an optimal increasing subsequence of $\vec{a}$. The second part computes an optimal increasing subsequence, but for convenience we print it out in reverse order. This program runs in time $O(n)$, so the entire algorithm runs in time $O(n^2)$.

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
        m ← i
    end if
end for
```

```
put a_m
while A(m) > 1 do
    i ← m − 1
    while not(a_i < a_m and A(i) = A(m) − 1) do
        i ← i − 1
    end while
    m ← i
    put a_m
end while
```