On presenting dynamic programming algorithms (Longest Increasing Subsequence)

Vassos Hadzilacos

When I present a dynamic programming algorithm in class I try to mimic the process by which one might go about discovering the algorithm: With your help, I analyze the problem at hand with an eye towards discovering its "optimal substructure" and use this analysis to determine the subproblems to solve and the recursive formula by which to compute these subproblems. In the process of doing so we also discuss the reason why the recursive formula is a correct way to compute the subproblems, often invoking a "cut-and-paste" argument that may be completely obvious or may require some explanation.

Although I hope that this type of presentation is pedagogically useful, it is not necessarily the clearest way to present the algorithm, if the goal is to ***describe*** it (and its correctness) to someone — such as the grader of your assignment — as opposed to illuminating how one might ***discover*** it, which is my goal in lecture.

After you have discovered the algorithm it may be clearer to use a dryer and more direct style, consisting of the steps listed below. I encourage you to use this style in presenting dynamic programming algorithms in your assignment solutions.

(1) State (very clearly!) the subproblems to solve.
(2) State how to compute the original problem given the solution to these subproblems.
(3) Give the recursive formula by which to compute the subproblems.
(4) Justify why the recursive formula correctly computes the subproblems. (This is the core argument for the algorithm's correctness; if it is done well, nothing else is necessary as the pseudocode is a straightforward transcription of this formula.)
(5) Describe the algorithm in pseudocode.
(6) Analyze the algorithm's running time.

The rest of this document illustrates this style of presentation. We will use as our example the very first dynamic programming algorithm we saw: computing (the length of) a longest increasing subsequence of an array $A[1..n]$ of numbers.

<u>Subproblems to solve:</u> For each i, $1 \le i \le n$, let

$$L(i) = \text{the length of a longest increasing subsequence of } A \textbf{\textit{ that ends in position }} i. \qquad (*)$$

<u>Solving the original problem:</u> The answer to our original problem is simply $\max\{L(i) : 1 \le i \le n\}$.

<u>Recursive formula to compute the subproblems:</u>

$$L(i) = \begin{cases} 1, & \text{if } A[j] \ge A[i], \text{ for all } 1 \le j < i \\ 1 + \max\{L(j) : 1 \le j < i \text{ and } A[j] < A[i]\}, & \text{otherwise} \end{cases} \qquad (\dagger)$$

<u>Justification why (†) correctly computes (*):</u> There are two cases:

Case 1. For every j, $1 \le j < i$, $A[j] \ge A[i]$. In this case, the longest increasing subsequence of A that ends in position i consists of just $A[i]$, and so it has length 1. So the formula (†) is correct in this case.

CASE 2. For some j, $1 \leq j < i$, $A[j] < A[i]$. Let S be a longest increasing subsequence of A that ends in position i. Therefore $S = S' \circ A[i]$, for some nonempty sequence S'.

S' is an increasing subsequence of A that ends at some j, $1 \leq j < i$, such that $A[j] < A[i]$. This is because, otherwise, S would not be an increasing subsequence of A (never mind a longest one).

Furthermore, S' is ***longest*** among all longest increasing subsequences of A that end at some position j, $1 \leq j < i$, such that $A[j] < A[i]$. For, if S' is not, then there is an increasing subsequence S'' of A that ends at some j, $1 \leq j < i$, such that $A[j] < A[i]$, and S'' is longer than S'. But then $S'' \circ A[i]$ is an increasing subsequence of A that ends at i that is longer than $S' \circ A[i] = S$, contradicting the definition of S.[1] Therefore, $L(i) = 1 + \max\{L(j) : 1 \leq j < i \text{ and } A[j] < A[i]\}$, and the formula (†) is correct in this case.

PSEUDOCODE: Our algorithm computes $L(1), L(2), \ldots, L(n)$, in this order, based on (†).

```
  LIS(A)
1 for i := 1 to n do
2     L(i) := 1
3     for j := 1 to i − 1 do
4         if A[j] < A[i] and L(j) + 1 > L(i) then L(i) := L(j) + 1
5 return max{L(i) : 1 ≤ i ≤ n}
```

RUNNING TIME ANALYSIS: The loop in lines 1-4 takes $\Theta(n^2)$ time and line 5 takes $\Theta(n)$ time, so the overall running time of the algorithm is $\Theta(n^2)$.

[1] This is a so-called "cut-and-paste" argument: We "cut" S' and "paste" S''. This type of argument is so standard that, ***in such simple situations***, you can simply say "By a cut-and-paste argument…". If the reasoning is more complex, you would have to explain more.