

1. Suppose we are given an array, A , of n positive numbers (or a string of n characters)

$$A = [a_1, a_2, \dots, a_n].$$

An **increasing subsequence** of A is a subsequence, $[a_{i_1}, a_{i_2}, \dots, a_{i_k}]$, of elements from A such that $i_j < i_{j+1}$ and $a_{i_j} < a_{i_{j+1}}$ for each $1 \leq j < k$. Notice that the elements in the increasing subsequence need not be contiguous in A .

Given an increasing subsequence of A , we can compute the length of this subsequence. The “longest increasing subsequence” problem is to find the maximum possible length over all possible increasing subsequences from the given array (or string) A .

Let us consider how to solve this problem using dynamic programming. We need to come up with a definition of a subproblem for which we can write a recurrence relation relating the solution of the subproblem to the solutions of other (smaller) subproblems. The most obvious choice for a subproblem is

$$L(j) = \text{length of the longest increasing subsequence between } a_1 \text{ and } a_j.$$

However, this is not a good definition for a subproblem. Suppose we know the solution to subproblem $L(j-1)$. Can we use that information to construct the solution to subproblem $L(j)$? Can we use element a_j to extend the solution to $L(j-1)$ into a solution to $L(j)$? The answer is no, because even though we know the length, $L(j-1)$, of the longest increasing subsequence between a_1 and a_{j-1} , we do not know at which element a_i that subsequence ended, so we don't know if a_j can be used to extend that subsequence. So we cannot write a recurrence relation relating the subproblem $L(j)$ to any of its (smaller) subproblems $L(i)$ where $i < j$.

Here is a more useful definition of a subproblem.

$$LE(j) = \text{length of the longest increasing subsequence that ends at } a_j.$$

This definition of a subproblem has the advantage that we can define a recurrence relation using it, but it has a slight disadvantage in that $LE(n)$ is *not* the solution of the problem we started out with (why?).

The idea behind the recurrence relation for $LE(j)$ is this. Given $LE(j-1)$, we know both the length of an increasing subsequence and where that subsequence ends. We can use a_j to increase the length of that subsequence if $a_j > a_{j-1}$, and the new increasing subsequence will have length $1 + LE(j-1)$. But even if we can extend the subsequence given by $LE(j-1)$, the resulting subsequence need not be the longest increasing subsequence that ends at a_j . It may be that we can extend some other, longer, increasing subsequence, $LE(i)$ where $i < j-1$ (and $a_i < a_j$). So to find the longest possible increasing subsequence that ends at a_j , we need to take a maximum over all of the sequences that a_j can extend. This leads to the following recurrence relation.

$$LE(j) = 1 + \max_{a_i < a_j} \{ LE(i) \}$$

where we use the convention that the maximum over an empty set is zero. (If $a_j \leq a_i$ for all $1 \leq i < j$, then the above maximum is over an empty set of indices. This can happen, for example, when the array A is sorted in decreasing order.)

The problem we want to solve is $L(n)$, where L still has the definition it was given above. We can define $L(n)$ in terms of the values of $LE(j)$. Since the solution to $L(n)$ must be a subsequence that ends somewhere, that maximizing subsequence must be a solution to one of the subproblems $LE(j)$ for some j between 1 and n . So

$$L(n) = \max_{1 \leq j \leq n} \{ LE(j) \}.$$

We can implement the recurrence relation for LE as a bottom-up (or top-down) dynamic programming solution. You should convince yourself that this would be an $\Theta(n^2)$ algorithm (because the calculation of each $LE(j)$ requires searching back through the array and looking at each element a_i for $1 \leq i < j$). The calculation of $L(n)$ from the table of $LE(j)$ values is a $\Theta(n)$ procedure. So the whole algorithm is $\Theta(n^2)$.

There is a $\Theta(n \lg(n))$ dynamic programming algorithm that solves this problem. It uses an even trickier definition of a subproblem than the one we used here. See *Introduction to Algorithms: A Creative Approach*, by Udi Manber, pages 167–169.

Notice that we have only provided the maximum *length* of the longest increasing subsequence. We have not calculated the location in the array of this maximal increasing subsequence, nor have we determined if there might be more than one increasing subsequence that gives this maximal length.