Guide to Dynamic Programming

Based on a handout by Tim Roughgarden. Thanks to Kostas Kollias, Andy Nguyen, Julie Tibshirani, and Sean Choi for their input!

Like greedy algorithms, dynamic programming algorithms can be deceptively simple. The algorithms, once written out, are often so straightforward that it's hard to believe they work correctly. Consequently, one of the challenges in writing dynamic programming algorithms is rigorously establishing their correctness.

Fortunately, dynamic programming proofs are often relatively straightforward and follow a standard pattern. This handout explores that pattern and gives guidelines about what we're looking for in a proof of correctness.

Dynamic Programming Proofs

Typically, dynamic programming algorithms are based on a recurrence relation involving the optimal solution, so the correctness proof will primarily focus on justifying why that recurrence relation is correct.

The general outline of a correctness proof for a dynamic programming algorithm is as following:

- **Define Subproblems**. Dynamic programming algorithms usually involve a recurrence involving some quantity OPT(k_1, …, k_n) over one or more variables (usually, these variables represent the size of the problem along some dimension). Define what this quantity represents and what the parameters mean. This might take the form "OPT(k) is the maximum number of people that can be covered by the first k cell towers" or "OPT(u, v, i) is the length of the shortest path from u to v of length at most i."
- **Write a Recurrence**. Now that you've defined your subproblems, you will need to write out a recurrence relation that defines OPT(k_1, …, k_n) in terms of some number of subproblems. Make sure that when you do this you include your base cases.
- **Prove the Recurrence is Correct**. Having written out your recurrence, you will need to prove it is correct. Typically, you would do so by going case-by-case and proving that each case is correct. In doing so, you will often use a "cut-and-paste" argument to show why the cases are correct.
- **Prove the Algorithm Evaluates the Recurrence**. Next, show that your algorithm actually evaluates the recurrence by showing that the table values match the value of OPT and that as you fill in the table, you never refer to a value that hasn't been computed yet. To be fully rigorous, you would probably need to prove this by induction *For the purposes of CS161, you do not need to go into this level of detail*. A few sentences should suffice here.
- **Prove the Algorithm is Correct**. Having shown that you've just evaluated the recurrence correctly, your algorithm will probably conclude with something like "return A[m, n]" or "return C[0]." Prove that this table value is the one that you actually want to read.

This might seem like a lot to do, but most of these steps are very short. The main challenge will be proving that the recurrence is correct, and most of the remaining steps are there either to set up this step or to show how that step leads to correctness.

Sample Problem: Longest Increasing Subsequence

The *longest increasing subsequence* (LIS) problem is a classic dynamic programing problem specified as follows. You are given an array n of values and want to find the longest subsequence of that array where the values are in strictly increasing order. For example, given an array holding ⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3⟩, one LIS is ⟨3, 4, 5, 9⟩. Another is ⟨1, 4, 5, 6⟩.

For the purposes of this problem, we'll assume all we care about is the *length* of the longest increasing subsequence, rather than the sequence itself.

Algorithm: Create a table DP of length n, where n is the length of array A. For $i = 1$ to n, set $DP[i] = \max\{1, \max_{j < i,\ A[j] < A[i]}\{1 + DP[j]\}\}$. (Here, assume that the max operator applied to an empty set produces $-\infty$.) Finally, return the maximum value in the DP array.

Correctness: For notational simplicity, let A[1, i] denote the first i elements of the array. Let OPT(i) denote the length of a LIS of A[1, i] that ends at position i. We claim that OPT(i) satisfies this recurrence:

$$\mathrm{OPT}(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\left\{ 1, \max_{j<i,\, A[j]<A[i]} \{1 + \mathrm{OPT}(j)\} \right\} & \text{otherwise} \end{cases}$$

Theorem: OPT(i) satisfies the above recurrence.

Proof: If $i = 0$, there is only one subsequence of the first 0 elements of the array, the empty sequence. Thus OPT(i) = 0. If $i \geq 1$, let S^* be a LIS of A[1, i]. So consider all but the last element of S^*; call this S. If S is empty, then S^* has length one, so OPT(i) = 1. Otherwise, S is nonempty, so S must end at some position j of the array. Since S^* is an increasing subsequence, we know $A[j] < A[i]$. Moreover, S must be a LIS of A[1, k] for some $k < i$ where $A[k] < A[i]$. If not, we could exchange S for a LIS of such a subarray, increasing the length of S^* and contradicting the fact that S^* is optimal. Thus, the length of S^* is given by the value $\max_{j < i,\ A[j] < A[i]}\{1 + \mathrm{OPT}(j)\}$. ■

Theorem: The algorithm returns the length of the longest increasing subsequence of A.

Proof: Our algorithm evaluates the recurrence OPT(i) bottom-up by storing OPT(i) in DP[i]. Note that since we evaluate DP[i] as i increases from 1 to n, all values for subproblems referenced by the recurrence for OPT(i) will have already been computed. At the end, our algorithm returns the maximum value in the DP array, which corresponds to the length of the longest LIS that ends somewhere in the array, which in turn is an LIS of the overall array A. ■

Runtime: We can implement the algorithm to run in time $O(n^2)$ as follows: on iteration i, scan across array indices $k = 1, 2, \ldots, i - 1$ and of the values seen that are less than A[i], choose the one with the highest corresponding DP[k] value (if one exists). Then, set DP[i] to the max of that value (if it exists), 1, and DP[$i - 1$]. This takes time $O(n)$, so across n iterations the work is $O(n^2)$. The final step of the algorithm (scanning over the array) can be implemented in time $O(n)$, so the total work done is $O(n^2)$.