A Practical Introduction to Dynamic Programming

Michael Levet

June 21, 2018

1 Introduction to Dynamic Programming

Dynamic programming is a powerful technique to solve computational problems, which have a recursive substructure and recurring subproblems. The idea is to solve these recursive subcases and store these solutions in a lookup table. When a solved recursive subcase is encountered, the existing solution is accessed using only a constant number of steps. A solution to the initial instance is constructed from the solutions to the sub-cases, typically in a bottom-up manner. Frequently, the computational problems of interest are optimization problems. Common examples include the Shortest Path Problem, the Rod-Cutting Problem, and the Game of Nim. We provide a practical exposition, introducing some examples amenable to the dynamic programming technique. The goal of this tutorial is that the readers can successfully apply the dynamic programming technique. To that end, this tutorial is not a complete treatment of the subject. In particular, issues of computational complexity are largely not discussed here. Readers interested in more advanced expositions should direct their attention to common algorithm analysis texts, such as Sedgewick and Wayne, or CLRS.

1.1 The Game of Nim

We consider the following two-person game, in which players alternate turns. The game begins with a pile of n (identical) stones. During a player's turn, they may remove either $1, 2$, or 3 stones from the pile. If a player cannot make a move, that player loses. This game is denoted as a $(1, 2, 3)$-Nim game, in light of the allowed moves of removing $1, 2$, or 3 stones. We define the game of Nim more formally.

Definition 1 (Nim)**.** Let $a_1, a_2, \ldots, a_k \in \mathbb{Z}^+$ be distinct, and let $n \in \mathbb{N}$. The game $(a_1, a_2, \ldots, a_k)$-Nim is a two player game, which is initialized with a pile of n stones. Players alternate removing stones from the pile, where the number of stones each player can remove on their turn lies in the set $\{a_1, a_2, \ldots, a_k\}$. A player loses if they cannot make a move on a given turn.

Remark: In $(1, 2, 3)$-Nim, a player loses if there are no stones left. However, in $(2, 3)$-Nim, a player loses if there are fewer than 2 stones left on the pile.

We restrict attention to $(1, 2, 3)$-Nim, with the goal of illustrating the dynamic programming technique to determine for which values of $i \in [n]$ Player 1 will win. Here, we assume both agents play optimally; that is, both players seek to win the game and are able to determine the best move to achieve their goal. We denote ℓ to indicate a loss, and w to denote a win.

(a) We begin by initializing our lookup table $T[0, \ldots, n]$ to store whether or not Player 1 will win, given a pile with i stones. Clearly, if $i = 0$, Player 1 loses. So $T[0] = \ell$. Similarly, if $i \in [3]$, then Player 1 can take all the stones and win in one turn. So $T[1] = T[2] = T[3] = w$.

(b) Now suppose there are $i = 4$ stones. No matter how many stones Player 1 takes (whether it be $1, 2$, or 3 stones), Player 2 takes the remaining stones. So Player 1 always loses. Thus, we set $T[4] := \ell$.

(c) Suppose there are $i = 5$ stones. Suppose Player 1 selects $j \in [3]$ stones. Now Player 2 is the first player in a smaller instance of $(1, 2, 3)$-Nim with $5 - j$ stones. Here, we begin to see the power of dynamic programming in constructing strategies. We have already computed whether Player 2 will win in the smaller instance of Nim with $5 - j$ stones, and so we can just look up the result in T. Observe that $T[5 - j] = \ell$ if and only if $j = 1$ (i.e., $T[4] = \ell$). So for $i = 5$, Player 1 starts by taking a single stone. Then Player 2 will lose. So $T[5] = w$.

(d) Using similar reasoning as in part (c), we have that $T[6] = T[7] = w$.

(e) Now what happens if there are $i = 8$ stones? Suppose that Player 1 takes $j \in [3]$ stones. So Player 2 is the first player in a smaller instance of $(1, 2, 3)$-Nim with $8 - j$ stones. For each $j \in [3]$, $T[8 - j] = w$. So Player 2 will always win. Thus, $T[8] = \ell$.

While we can continue building the lookup table, it may be more insightful to look at the entries already present. Observe that $T[0], T[4]$, and $T[8]$ are the only entries with ℓ. This leads to the following observation.

Proposition 1.1. *In $(1, 2, 3)$-Nim with n stones in the pile, Player 1 has a winning strategy if and only if n is not a multiple of 4.*

Proof. The proof is by strong induction on $n \in \mathbb{N}$. We have the following base cases:

- **Case:** Suppose $n = 0$. Player 1 has no available moves, and so Player 1 loses.
- **Case:** Suppose $n \in [3]$. Player 1 takes all available stones. So Player 2 has no moves and loses. Thus, Player 1 wins.

Now fix $k \geq 4$, and suppose that the proposition holds for all $0 \leq n \leq k$. We prove true for the $k + 1$ case. have the following cases:

- **Case:** Suppose that $k+1$ is not a multiple of 4. By the Division Algorithm, we may write $k+1 = 4q+r$ for some $r \in [3]$. We show that it is a winning strategy for Player 1 to take r stones on their first turn. After Player 1 takes r stones, Player 2 takes their turn with $k + 1 - r = 4q$ stones remaining. Observe that Player 2 is the first player in a smaller instance of $(1, 2, 3)$-Nim with $4q$ stones. By the Inductive Hypothesis, Player 2 has no winning strategy, as $4q$ is a multiple of 4. So Player 1 has a winning strategy, as claimed.
- **Case:** Suppose that $k+1$ is a multiple of 4. For any $i \in [3]$, $k+1-i$ is not a multiple of 4. Suppose Player 1 removes i stones from the pile. Then Player 2 is the first player in a smaller instance of $(1, 2, 3)$-Nim with $k + 1 - i$ stones. As $k + 1 - i \leq k$ and $k + 1 - i$ is not a multiple of 4, we have by the inductive hypothesis that Player 2 has a winning strategy. Thus, Player 1 does not have a winning strategy, as claimed.

The result follows by induction. □

Remark: When working with instances of Nim, it is helpful to employ dynamic programming with the goal of determining the *period* of the game, or the length of the pattern of wins and losses that repeat within the lookup table. Once this pattern is ascertained, we may appeal to the pattern to decide in constant time who wins the game. More exposition and generalizations of Nim are discussed in Combinatorial Game Theory, and we direct interested readers to look there for more in-depth exposition on Nim.

1.2 Rod-Cutting Problem

In this section, we examine the Rod-Cutting Problem. Let us consider a motivating example. Suppose we have a rod of length 5, which can be cut into smaller pieces of lengths $1, 2, 3$, or 4. These smaller rods can thenfurther be cut into smaller pieces. Now suppose that we can sell rods of length 1 for \$1, which we denote $p_1 = 1$. Similarly, suppose that the prices for rods of length $2, \ldots, 5$ are given by $p_2 = 4, p_3 = 7, p_4 = 8$, and $p_5 = 9$ respectively. We make two key assumptions: we will sell all the smaller rods, regardless of the cuts; and that each cut is free. Under these assumptions, how should the rod be cut to maximize the profit? We note the following cuts and the corresponding profits.

- If the rod is cut into five pieces of length 1, we stand to make $5 \cdot p_1 = \$5$.
- If the rod is cut into one piece of length 2 and one piece of length 3, we stand to make $p_2+p_3 = 4+7 = \$11$.
- If the rod is cut into two pieces of length 2 and one piece of length 1, we stand to make $2 \cdot p_2 + p_1 = 8 + 1 = \$9$.

Out of the above options, cutting the rod into one piece of length 2 and one rod of length 3 is the most profitable. Of course, there are other possible cuts not listed above, such as cutting the rod into one piece of length 1 and one piece of length 4. The goal is to determine the most profitable cut. The Rod-Cutting Problem is formalized as follows.

Definition 2 (Rod-Cutting Problem)**.**

- `Instance:` Let $n \in \mathbb{N}$ be the length of the rod, and let $p_1, p_2, \ldots, p_n$ be non-negative real numbers. Here, p_i is the price of a length i rod.
- `Solution:` The maximum revenue, which we denote r_n, obtained by cutting the rod into smaller pieces of integer lengths and selling the smaller rods.

Intuitively, the maximum revenue is determined by examining the revenues for the subdivisions and taking the largest. Mathematically, this amounts to the following expression:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \tag{1}$$

We begin by working through an example of utilizing dynamic programming to determine the maximum profit.

Example 1. Suppose we have a rod of length 5, with prices $p_1 = 1, p_2 = 4, p_3 = 7, p_4 = 8, p_5 = 9$. We proceed as follows.

(a) Initialize a lookup table $T[1, \ldots, 5]$. Now for a rod of length 1, there is only one price: p_1. So we set $T[1] := p_1$.

(b) Now consider a rod of length 2. There are two options: either don't cut the rod, or cut the rod into two smaller pieces of length 1. Here, $p_2 = 4$ represents the case in which no cuts to a rod of length 2. Now suppose instead we cut the rod up into two smaller pieces each of length 1. We know that the maximum profit for a rod of length 1 is $r_1 = T[1] = 1$. So the profit of cutting the rod into two smaller pieces each of length 1 is $2r_1 = 2$. Now $r_2 = \max(p_2, 2r_1) = 4$, so we set $T[2] = 4$.

(c) Now consider a rod of length 3. Here, we have more options: we can leave the rod untouched, we can divide the rod into smaller pieces of length 1 or length 2; or we can divide the rod into three pieces of length 1. If we do not divide the rod into smaller pieces, the profit is $p_3 = 7$. Now suppose we divide the rod up into smaller pieces of length 2 and length 1. We may now keep this configuration, or further divide the rod of length 2 into two rods each of length 1, as discussed in the previous bullet point. Rather than re-solving this problem, we can simply look up the maximum profit for a rod of length 2 in the lookup table. Recall that $r_2 = T[2] = 4$ and $r_1 = 1$. So the profit from cutting the rod into smaller pieces of length 2 and length 1 is $r_2 + r_1 = 5$. Now $r_3 = \max(p_3, r_2 + r_1) = 7$, so we set $T[3] = 7$.

(d) Now consider a rod of length 4. We have the following options for the first cut: leave the rod uncut, in which we stand to make profit $p_4 = 8$; cut the rod into smaller pieces of length 1 and length 3; or cut the rod into two smaller rods, each of length 2. Consider the case in which we cut the rod up into smaller pieces of length 1 and length 3. The natural, though inefficient, approach here is to consider all the ways in which we could cut up the rod of legnth 3. It turns out that we don't need to do this, as the maximum revenue attainable from a rod of length 3 was found in the previous bullet point. This is the power of dynamic programming: once a solution to a smaller problem is found, we simply look it up rather than re-solving the smaller problem. Similarly, we can look up the maximum profit for a rod of length 2. So given our cases, we have the following possible profits:

- The uncut rod of length 4 will result in profit $p_4 = 8$.
- The rod cut into pieces of length 3 and length 1 will result in profit $r_3 + r_1 = T[3] + T[1] = 7 + 1 = 8$.
- The rod cut into two pieces, each of length 2, will result in profit $2r_2 = 2 \cdot T[2] = 2 \cdot 4 = 8$.

So $T[4] = \max(8, 8, 8) = 8$.

(e) Finally, consider our original rod of length 5. We have the following possible initial cuts:

- We can leave the rod uncut, in which case we will make profit $p_5 = 9$.
- We can cut the rod into one piece of length 4 and one piece of length 1. The maximum revenue attainable by cutting up a rod of length 4 was determined already. So we can simply look up this solution in $T[4]$. Thus, the profit in this case is $r_4 + r_1 = T[4] + T[1] = 8 + 1 = 9$.
- We can cut up the rod into one piece of length 3 and one piece of length 2. By similar argument as above, we may simply look up the maximum revenues attainable from a rod of length 3 and a rod of length 2. So our profit is $r_3 + r_2 = T[3] + T[2] = 7 + 4 = 11$.

So $r_5 = \max(9, 9, 11) = 11$. Thus, we set $T[5] = 11$.

We conclude that we stand to make \$11 from a rod of length 5.

While the expression (1) may not seem insightful, it in fact provides an algorithm to compute r_n. Example 1 provides a tangible example of this algorithm. The goal now is to generalize the algorithm from Example 1 to work for any rod of positive integer length any list of prices. We proceed as follows.

(a) Initialize the lookup table $T[1, \ldots, n]$, and set $T[1] := p_1$.

(b) We set $T[2] := \max(p_2, 2r_1)$. Here, p_2 represents the case in which no cuts to a rod of length 2, and $2r_1$ represents the case in which a rod of length 2 is cut into two rods each of length 1. We note that $r_1 = T[1] = p_1$.

(c) We set $T[3] := \max(p_3, r_1 + r_2)$. Now $r_1 = T[1]$, and $r_2 = T[2]$. We have already solved the rod cutting problem for a length 2 rod, so we simply look up r_2 in the table T rather than re-solving the problem.

(d) $T[4] := \max(p_4, r_1 + r_3, 2r_2)$. As we have already computed r_1, r_2, r_3, we may look up their respective values in T rather than re-computing these values.

Continuing in this manner, we compute r_n, which is the value in $T[n]$ after the algorithm terminates.

Remark: This algorithm only provides the maximum revenue. It does not tell us how to achieve that result. As an exercise, modify the algorithm to produce an optimal set of rod cuts.

1.3 Longest Common Subsequence Problem

Solutions to both the Rod-Cutting Problem and Nim utilized dynamic programming techniques, where the lookup table was one-dimensional. In this section, we introduce the Longest Common Subsequence Problem, which is also amenable to the dynamic programming technique. However, unlike the Rod-Cutting Problem and Nim, the lookup table for the Longest Common Subsequence Problem is a two-dimensional table rather than a one-dimensional array. The purpose of this section is to illustrate the usage of multidimensional lookup tables in dynamic programming problems. To this end, the Longest Common Subsequence Problem serves as a tangible example. We begin by formalizing the Longest Common Subsequence Problem.

Definition 3 (Subsequence)**.** Let Σ be a finite set, which we refer to as an *alphabet*. Let $\omega \in \Sigma^n$. We say that $\psi \in \Sigma^m$ is a subsequence of ω if there exists a strictly increasing sequence of indices $(i_1, i_2, \ldots, i_m)$ such that $\omega_{i_k} = \psi_k$ for all $k \in [m]$.

Example 2. Let $\omega = (A, B, C, B, D, A, B)$, and let $\psi = (A, C, D, B)$. Consider the sequence of indices $(1, 3, 5, 7)$. So $\psi_1 = \omega_1$, $\psi_2 = \omega_3, \psi_3 = \omega_5$, and $\psi_4 = \omega_7$. Thus, ψ is a subsequence of ω.

Definition 4 (Common Subsequence)**.** Let Σ be an alphabet. Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. We say that $\psi \in \Sigma^\ell$ is a *common subsequence* of ω and τ if: ψ is a subsequence of ω, and ψ is a subsequence of τ. Note that ψ does not have to appear as a subsequence in the same position in both ω and τ.

Example 3. Let $\omega = (0, 2, 1, 2, 3, 0, 1)$ and $\tau = (2, 3, 1, 0, 2, 0)$. The sequence $(2, 1, 0)$ is a subsequence of both ω and τ. Here, $(2, 1, 0)$ appears in ω at the indices $(2, 3, 6)$, and $(2, 1, 0)$ appears in τ at the indices $(1, 3, 4)$.

Definition 5 (Longest Common Subsequence Problem (LCS))**.**

- `Instance:` Let Σ be an alphabet, and let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences.
- `Solution:` A sequence ψ that is common to both ω and τ; and for any other common subsequence σ of ω and τ, $|\sigma| \leq |\psi|$.

The naïve approach to solving LCS is enumerating all the possible subsequences of X and Y, and recording the longest. Without loss of generality, suppose that $m \leq n$. So there are 2^m possible index sequences to check, which correspond bijectively to subsequences of Y. So for large sequences, the brute force and ignorance solution is not a practical solution. The dynamic programming approach provides a linear time algorithm instead.

Dynamic programming works best when optimal solutions to subproblems can be used to construct an optimal solution to the original instance. We first show that LCS exhibits this property.

Theorem 1.1. *Let Σ be an alphabet, and let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. Let $\psi \in \Sigma^k$ be a longest common subsequence of ω and τ. The following hold:*

(a) If $\omega_n = \tau_m$, then $\psi_k = \omega_n = \tau_n$ and $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$.

(b) If $\omega_n \neq \tau_m$ and $\psi_k \neq \omega_n$, then ψ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ. Similarly, if $\omega_n \neq \tau_m$ and $\psi_k \neq \tau_m$, then ψ is a longest common subsequence of ω and $\tau[1, \ldots, m-1]$.

Proof.

(a) Let σ be a common subsequence of ω and τ whose last digit does not correspond to the last instance of the character $\omega_n = \tau_m$ in ω and τ. Then σ can be augmented by appending the character $\omega_n = \tau_m$. So every longest common subsequence of ω and τ has last character $\omega_n = \tau_m$.

We now show that $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. Observe that $\psi[1, \ldots, k-1]$ is a common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. Suppose to the contrary that there exists a longest common subsequence σ of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$, with $|\sigma| > k$. Then σ can be augmented with $\omega_n = \tau_m$ to obtain a common subsequence of ω and τ. This contradicts the assumption that any longest common subsequence of ω and τ has length k. So $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$.

(b) Suppose that $\omega_n \neq \tau_m$. Now suppose that $\psi_k \neq \omega_n$. We show that ψ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ, by contradiction. Let σ be a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ of length $|\sigma| > k$. Clearly, σ is a common subsequence of ω and τ. Now $|\sigma| > |\psi| = k$, contradicting the assumption that ψ was a longest common subsequence of ω and τ. So ψ is a longest common subsequence of ω and τ. Interchanging the roles of ω and τ, we obtain that: if $\omega_n \neq \tau_m$ and $\psi_k \neq \tau_m$, then ψ is a longest common subsequence of ω and $\tau[1, \ldots, m-1]$.

□

Theorem 1.1 provides the insights necessary for designing a dynamic programming algorithm to solve LCS. Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. If $\omega_n = \tau_m$, we record the last character and examine the smaller LCS instance with $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. If $\omega_n \neq \tau_m$. Otherwise, we need to find the longest common subsequences of ω and $\tau[1, \ldots, m-1]$; and $\omega[1, \ldots, n]$ and τ. These observations yield a natural recurrence to compute the length of the longest common subsequence for a pair of strings:

$$\ell[i,j] = \begin{cases} 0 : & i = 0 \text{ or } j = 0, \\ \ell[i-1, j-1] + 1 : & i, j > 0 \text{ and } x_i = y_j, \\ \max(\ell[i-1, j], \ell[i, j-1]) : & i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Using the recurrence $\ell[i, j]$ as a template, we design an explicit dynamic programming algorithm. We proceed as follows.

(a) Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be our input sequences. We initialize a lookup table $T[0, \ldots, n][0, \ldots, m]$ to be a two-dimensional array, where each cell stores:

- A natural number corresponding to the length of a longest common subsequence; and
- A pointer to another cell in the lookup table, which corresponds to the optimal subproblem as specified in Theorem 1.1.

Now recall that if either of the input sequences have length 0, the length of the longest common subsequence is 0. Therefore, we set $T[i][0] = 0$ and $T[0][j] = 0$ for all $i \in [n]$ and all $j \in [m]$. While our original input sequences may not have length 0, sequences we encounter in subproblems may indeed have length 0.

(b) We now proceed to fill in the remaining cells in a bottom up manner, row-by-row. Each row is filled left-to-right. The cells $T[i][j]$ are filled as follows.

- **Case 1:** Suppose $\omega_i = \tau_j$. By Theorem 1.1, any longest common subsequence ψ of $\omega[1, \ldots, i]$ and $\tau[1, \ldots, j]$ ends with $\omega_i = \tau_j$. Furthermore, $\psi[1, \ldots, |\psi| - 1]$ is a longest common subsequence of $\omega[1, \ldots, i-1]$ and $\tau[1, \ldots, j-1]$. So we take the following actions:

- Set $T[i][j]$.length $= T[i-1][j-1]$.length $+ 1$; and
- Set $T[i][j]$.subproblem $= T[i-1][j-1]$.

- **Case 2:** Suppose $\omega_i \neq \tau_j$. Theorem 1.1 tells us that we need to consider the two subproblems, whose solutions (or at least, their optimal lengths) are stored in: $T[i-1][j]$ and $T[i][j-1]$, respectively. If $T[i-1][j]$.length $\geq T[i][j-1]$.length, we set:
  - $T[i][j]$.length $= T[i-1][j]$.length
  - $T[i][j]$.subproblem $= T[i-1][j]$.

  Otherwise, we set:
  - $T[i][j]$.length $= T[i][j-1]$.length
  - $T[i][j]$.subproblem $= T[i][j-1]$.

In order to construct a longest common subsequence from the lookup table T, we start at $T[n][m]$ and follow the pointers to the subproblem. Each time some $T[i][j]$ points to $T[i-1][j-1]$ as a subproblem, we prepend the character $\omega_i = \tau_j$ to the front of the longest common subsequence. We stop once the currently visited cell has no pointer to a subproblem.

Example 4. Let $\omega = (A, B, C)$ and $\tau = (B, A, C, B, D)$. By inspection, it is easy to see that any longest common subsequence of ω and τ has length 2. In particular, $(A, B), (B, C)$, and (A, C) are all longest common subsequences of ω and τ. We work through the dynamic programming algorithm to explicitly find a longest common subsequence.

(a) We begin by initializing a 4×6 lookup table $T[0, \ldots, 3][0, \ldots, 5]$, and filling the first row and column with 0's. So we have:

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0					
B	0					
C	0					

(b) We now fill Row 1.

- Consider $T[1][1]$. Observe that $\omega_1 = A$ and $\tau_1 = A$ are different. So $T[1][1]$.length is the maximum of $T[1][0]$.length $= 0$ and $T[0][1]$.length $= 0$. Thus, $T[1][1]$.length $= 0$. By Case 2 of the algorithm, $T[1][1]$.subproblem points to $T[1][0]$.
- Consider $T[1][2]$. Observe that $\omega_1 = \tau_2$. So $T[1][2]$.length $= T[0][1]$.length $+ 1 = 1$, and $T[1][2]$.subproblem points to $T[0][1]$.
- Consider $T[1][3]$. Observe that $\omega_1 = A$ and $\tau_3 = C$ are different. So $T[1][3]$.length is the maximum of $T[0][3]$.length $= 0$ and $T[1][2]$.length $= 1$. So $T[1][3]$.length $= 1$, and $T[1][3]$.subproblem points to $T[1][2]$.
- Consider $T[1][4]$. Observe that $\omega_1 = A$ and $\omega_4 = B$ are different. By similar argument as for $T[1][1]$ and $T[1][3]$, we set $T[1][4]$.length $= 1$ and $T[1][4]$.subproblem to point to $T[1][3]$.
- Consider $T[1][5]$. By similar argument as for $T[1][4]$, $T[1][5]$.length $= 1$ and $T[1][5]$.subproblem points to $T[1][4]$.

The updated lookup table is as follows:

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0	$\leftarrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$
B	0					
C	0					

(c) We next fill Rows 2-3, omitting the detailed explanation associated with filling Row 1. The completed lookup table is as follows.

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0	$\leftarrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$
B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 2$	$\leftarrow 2$
C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\leftarrow 2$

(d) Finally, we construct a longest common subsequence of ω and τ from the lookup table. We start at $T[3][5]$ and follow the arrows, prepending the character at the given index every time we see $\nwarrow$. So we have the sequence:

$$T[3][5] \rightarrow T[3][4] \rightarrow T[3][3] \text{ (Record C)}$$
$$\rightarrow T[2][2] \rightarrow T[2][1] \text{ (Record B)}$$
$$\rightarrow T[1][0].$$

After which, we stop, as $T[1][0]$ does not reference any subproblems. So our longest common subsequence is (B, C), which we identified at the start of this example.