Lecture 14: Dynamic Programming

The main subject of this lecture is the longest common subsequence (LCS) problem. Our textbook covers this topic very well. So I will only mention a few things.

For the subproblem $LCS(i, j)$, we look at the two cases: $X[i] = Y[j]$ and $X[i] \neq Y[j]$. When $X[i] = Y[j]$, I explained in class that it is always safe to include $X[i]$ in the LCS of $X[1..i]$ and $Y[1..j]$. Make sure you understand why. When $X[i] \neq Y[j]$, clearly either $X[i]$ is not included or $Y[j]$ is not included in the LCS. And one of these two must happen. We then take the better of the two choices.

During the lecture, I described how the LCS dynamic programming table is filled in, and how to find the LCS by trace-back. Read the textbook if you have doubts. You should be familiar with the track back after working on the two homework problems.

We now discuss the coin change problem. As we mentioned before, the simple greedy algorithm works for some instances of this problem, but does not work for other situations. We now apply dynamic programming to this problem. Let us say we have $k$ distinct coins, ordered by value in the decreasing order: $d_1, d_2, \ldots, d_k = 1$. Here $d_i > d_{i+1}$. For example, we may have $d_1 = 25, d_2 = 10, d_3 = 5, d_4 = 1$. Our goal is to make changes for $n$ cents using the smallest number of coins. We assume there are unlimited supply of each type of coins.

1. We define the subproblem: $C[i]$ is equal to the minimum number of coins for $i$ cents.
2. Initialization: $C[j] = \infty$ if $j < 0$, and $C[0] = 0$.
3. Recurrence: $C[i] = 1 + \min_{x=1}^{k} (C[j - d_x])$.
4. Based on this recurrence, it is straightforward to write down the algorithm (omitted).
5. Time: $O(nk)$. This is because there are $n$ iterations, each taking $O(k)$ time.
6. How to find the opt solution? Backtrack again. Omit: find out how you get there by using which coin at that step, and recursively search.

Lecture 15: Dynamic Programming

In this lecture, we continue our discussion of dynamic programming. We discussed two problems: 0-1 Knapsack, matrix chain multiplication and weighted activity selection.

Recall that in the 0-1 Knapsack problem, we have $n$ items, each weight $w_i$ (which is an integer) pound and worth $v_i$ dollars. The goal is find a subset of items as valuable as possible but no more than $W$ pounds (here $W$ is a given integer). For each item, we must either take or not take and thus is called 0-1 Knapsack.

Suppose the optimal solution contains $k$ items, $b_1, b_2, \ldots, b_k$. Suppose we remove $b_k$, then we have the following property: the remaining items, $b_1, b_2, \ldots, b_{k-1}$ must the most valuable items under the constraints that total weight is no more than $W - w_{b_k}$. Otherwise, we have a contradiction that $b_1, b_2, \ldots, b_k$ are the most valuable items with no more than $W$ total weight.

Now we define $M[i, w]$ as the highest value (in dollar) when we choose items from 1 to $i$, and total weight is no more than $w$. Clearly, the solution to our Knapsack problem is simply $M[n, W]$. It is easy to see that $M[0, w] = 0$, and $M[i, 0] = 0$ for all $i/w$. When $i \geq 1$ and $w > 0$, we have: $M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w])$ if $w_i < w$, and otherwise $M[i, w] = M[i-1, w]$. This is because we either take item $i$ or not. If we take item $i$, then the remaining capacity of Knapsack is reduced by the weight of item $i$. If not, the capacity remains unchanged. In either case, we should pack as valuable as we can using the remaining items.

1: \textbf{for} $w = 0$ to $W$ \textbf{do} \\
2: \hspace{1em} $M[0, w] = 0$ \\
3: \textbf{end for} \\
4: \textbf{for} $i = 1$ to $n$ \textbf{do} \\
5: \hspace{1em} $M[i, 0] = 0$ \\
6: \hspace{1em} \textbf{for} $w = 1$ to $W$ \textbf{do}

CSE 3500 — Algorithms and Complexity — Yufeng Wu — Fall 2011
Lecture Notes ——- Week 8
if \( w_i \leq w \) then
\[
M[i, w] = \text{MAX}(M[i - 1, w - w_i] + v_i, M[i - 1, w])
\]
else
\[
M[i, w] = M[i - 1, w]
\]
end if
end for
end for
return \( M[n, W] \).

Running time: we have \( O(nW) \) cells in \( M \) array, each taking \( O(1) \) to compute. So the total time is \( O(nW) \).

Is this a polynomial-time algorithm? Not quite but it works well when \( W \) is relatively small.

The next problem is matrix chain multiplication. Again, this is a problem that is well explained in the textbook. I want to emphasize several things. First, finding the subproblems is often one of the most important aspects of using dynamic programming. For this problem, we define \( M[i, j] \) (where \( i \leq j \)) as the smallest amount of computation needed to multiple matrices \( A_i \ldots A_j \). This is natural: maybe the optimal solution will put parenthesis right before \( A_i \) and after \( A_j \). Of course, we are not sure, but the key idea of DP is to compute the results for all the subproblems and then figure out the overall solution from these subproblems. I will skip the rest of details (since the textbook explains well). I suggest you to experiment with small examples to understand why the proposed DP algorithm works.

The last problem is the weighted activity selection problem, which generalizes the activity selection problem discussed in the greedy algorithm session. It is easy to see that in this case, the previous greedy solution “prefer earlier ending time” no longer gives optimal solution. Now we use dynamic programming. Assume we have \( n \) activities, sorted by their ending time. Each activity \( A_i \) has a weight \( w_i \). We define \( W[i] \) as the maximum weight one can get by picking from activities \( A_1 \) to \( A_i \). The key observation is that we should look at activity \( A_i \). If we do not select \( A_i \), then \( W[i] = W[i - 1] \). If we do select \( A_i \), then \( W[i] = w_i + W[p(i)] \), where \( p(i) \) is the largest index of activities that is compatible with \( A_i \). Then \( W[i] = \text{max}(W[i - 1], w_i + W[p(i)]) \).