Lecture 14: Dynamic Programming

We discussed two example problems: the longest common subsequence (LCS) and the general coin change problem. The LCS problem is well explained in the textbook, and I will skip it here. Read the section in the textbook if you have questions about the LCS problem. Here are a few key points.

- We use a two dimensional array: $LCS[i, j]$, which represents the length of LCS of two prefixes $X[1..i]$ and $Y[1..j]$. By this definition, the length of LCS of $X$ and $Y$ is equal to $LCS[n, m]$, where $n$ and $m$ are the length of $X$ and $Y$ respectively. In class, a student suggested the solution can be found by finding the largest value with $LCS[i, j]$. This is correct, but checking the entire $LCS$ array is not necessary because $LCS[n, m]$ is the largest among $LCS[i, j]$ (why?).

- A key to dynamic programming is how to come up with recurrences of the subproblems. The recurrence allows you to compute the value of a subproblem using the pre-computed values of smaller subproblems. This often involves analyzing the situation to see what you can do to some specific parts of the subproblems. Often you have multiple choices, and you will just take the most desirable one. In the LCS example, we consider what happens to the last two characters. If these two characters are the same, then they must be in the LCS; otherwise, at least one will not be in the LCS. In either case, this allows us to relate the current subproblems to some smaller subproblems.

- 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: $d_1, d_2, ..., d_k = 1$. 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: $C[i] = 1 + \min_{x=1}^{k}(C[i - d_x])$.
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, continued

In this lecture, we continue our discussion of dynamic programming. We discussed two problems: 0-1 Knapsack.

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 \( W - w_k \). 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:

\[
M[i, w] = \max(M[i - 1, w - w_i] + v_i, M[i - 1, w])
\]

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.

```plaintext
1: for w = 0 to W do
2:     M[0, w] = 0
3: end for
4: for i = 1 to n do
5:     M[i, 0] = 0
6:     for w = 1 to W do
7:         if \( w_i \leq w \) then
8:             M[i, w] = \max(M[i - 1, w - w_i] + v_i, M[i - 1, w])
9:         else
10:            M[i, w] = M[i - 1, w]
11:        end if
12:    end for
13: end for
14: 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 multiply 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.

Finally, I want to say a few words on dynamic programming. The key to make dynamic programming work is the optimal substructure property: construct optimal solution for a problem with optimal solution of its subproblems. The subproblem needs to optimal (which does not impact on the rest of solution). There is a big difference between DP and greedy algorithms (which also with optimal substructure): the greedy algorithm is a top-down approach, while DP is bottom up. That is, the greedy algorithm makes a decision, and then move on to the resulting subproblem. Be careful about optimal substructure property. This is the example given in the textbook. Consider a directed graph \( G(V, E) \). We want to find the longest simple path (can not repeat edges). This problem does not have optimal substructure property. See Figure 15.6 (p. 382) for an example.