12.1 Introduction

Today we're going to do a couple more examples of dynamic programming. While the particular problems that we're going to talk about are important and very nice, we're spending a class on them not so much because of their inherent interest, but to see some more examples of dynamic programming. Since last class I mentioned the usefulness of dynamic programming in string algorithms, we're first going to talk about the *Longest Common Subsequence (LCS)* problem. Then, since we've spent some time recently on binary search trees, we're going to talk about the *Optimal Binary Search Tree* problem.

12.2 Longest Common Subsequence

12.2.1 Definitions

While there are many notions of similarity between strings, and many problems that we would like to optimize over strings, a natural problem (and notion of similarity) is the Longest Common Subsequence.

Definition 12.2.1 *Given a sequence $X = (x_1, x_2, \ldots, x_m)$ (where each x_i is an element of some alphabet, e.g. $\{0, 1\}$ or the English alphabet), another sequence $Z = (z_1, \ldots, z_k)$ is a* subsequence *of X if there exists a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ of indices of X such that $x_{i_j} = z_j$ for all $j \in \{1, 2, \ldots, k\}$.*

Less formally, Z is a subsequence of X if we can find Z in X, where we are allowed to skip elements of X. In a *substring*, on the other hand, we are not allowed to skip elements of X – slightly more formally, in a substring we require $i_j = i_{j-1} + 1$ for all $j \in \{2, 3, \ldots, k\}$. So, for example, (B, C, D, B) is a subsequence of (A, B, C, B, D, A, B) but is not a substring. For today, we're only going to be concerned with subsequences.

Given two sequences X and Y, we say that Z is a *common subsequence* if Z is a subsequence of X and Z is a subsequence of Y. In the Longest Common Subsequence problem, we are given two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots y_n)$ and wish to find the common subsequence of maximum length.

12.2.2 Dynamic programming algorithm

We first need to figure out what our subproblems should be, based on what kind of optimal substructure result we can prove. Let $X_i = (x_1, \ldots, x_i)$ be the length i prefix of X, and similarly let Y_i be the length i prefix of Y. Let $OPT(i, j)$ be the longest common subsequence of X_i and Y_j (so $OPT(m, n)$ is the subsequence that we are actually looking for). Then we can relate these subproblems with the following theorem. Informally, it says that if the last two entries of X_i and

Y_j match up then they must be the last element of the LCS, and if they do not match up then the LCS must be the LCS of some prefixes.

Theorem 12.2.2 *Let $Z = (z_1, \ldots, z_k)$ be an LCS of X_i and Y_j (so $Z = OPT(i, j)$).*

1. *If $x_i = y_j$, then $z_k = x_i = y_j$ and $Z_{k-1} = OPT(i-1, j-i)$*
2. *If $x_i \neq y_j$, then if $z_k \neq x_i$, $Z = OPT(i-1, j)$*
3. *If $x_i \neq y_j$, then if $z_k \neq y_j$, $Z = OPT(i, j-1)$*

Proof: We consider each of the three cases.

1. Suppose that $z_k \neq x_i$. Then we can add an extra element to z by setting $z_{k+1} = x_i = y_j$ and get a common subsequence of length $k+1$, violating our assumption that Z is an LCS of X_i and Y_j. Thus $z_k = x_i = y_j$. Thus Z_{k-1} is a common subsequence of X_{i-1} and Y_{j-1}; we just need to show that it is the longest common subsequence. Suppose there is a common subsequence W of length more than $k-1$. Then by appending $x_i = y_j = z_k$ to W, we get an LCS of X_i and Y_j of length more than k, again contradicting our assumption.
2. If $x_i \neq y_j$ then it cannot be the case that both x_i and y_j are equal to z_k. If $x_j \neq z_k$, then the index i_k which places z_k into X must be less than i. Thus Z is an LCS of X_{i-1} and Y_j.
3. Symmetric to previous case.

■

With this theorem in hand, we can naturally write a recurrence relation/recursive algorithm. Mathematically, this is given by the relation

$$OPT(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0, \\ OPT(i-1, j-1) \circ x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(OPT(i, j-1), OPT(i-1, j)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{12.2.1}$$

If we implement this using the obvious recursive algorithm, then it's not hard to construct examples that take exponential time (good exercise to do at home!). But we can notice that there are only nm different subproblems, so dynamic programming gives a polynomial-time solution! As always, we can write either a top-down memoized algorithm or a bottom-up DP algorithm. There's actually a somewhat interesting thing to note about the bottom-up algorithm: since we have a two-dimensional table, we have to make sure that the iteration we use makes sense. That is, when we try to fill in a table entry, we need to make sure that the table entries we use are already filled in. This was obvious in the interval scheduling problem, since when we filled in $M[j]$ we just needed the values of $M[i]$ for $i < j$. But now the table is two-dimensional, so what order should we fill it in? It turns out that, as long as we initialize appropriately, we can iterate through one parameter and then the other in a straightforward way.

```
LCS(X,Y) {
  for(i = 0 to m) M[i,0] = 0;
  for(j = 0 to n) M[0,j] = 0;
  for(i = 1 to m) {
    for(j = 1 to n) {
      if(x_i = y_j)
        M[i,j] = 1 + M[i-1,j-1];
      else
        M[i,j] = max(M[i,j-1], M[i-1,j]);
    }
  }
  return M[m,n];
}
```

Clearly the running time of this algorithm is $O(mn)$, since it is just two nested for loops. To prove correctness, we claim that $M[i,j] = |OPT(i,j)|$. Note that this is clearly true by (12.2.1) for the case where $i = 0$ or $j = 0$. For every other case, we will prove this by induction on $i + j$. For the base case, when $i + j = 0$ then $i = j = 0$, and by construction $M[0,0] = 0 = |OPT(0,0)|$. For the inductive step, consider $M[i,j]$. If $x_i = y_j$, then the algorithm sets

$$M[i,j] = 1 + M[i-1,j-1] = 1 + |OPT(i-1,j-1)| = |OPT(i,j)|,$$

where the second equality is by induction and the final equality is by (12.2.1). Similarly, if $x_i \neq y_j$, then the algorithm sets

$$M[i,j] = \max(M[i,j-1], M[i-1,j]) = \max(|OPT(i,j-1)|, |OPT(i-1,j)|) = |OPT(i,j)|,$$

where the second equality is by induction and the third is by (12.2.1).

As with weighted interval scheduling, once we've filled in the table M with the appropriate lengths, we can do a second pass to figure out the subsequence itself.

12.3 Optimal Binary Trees

12.3.1 Definitions

The next example of dynamic programming that we will consider is the problem of constructing an optimal binary search tree. In some applications, the keys are fixed and we also have a good idea of approximately how often each key is accessed. For example, if we are building a program to translate from English to some other language, we might store each English word as a key in a binary search tree, and the translation of the word as the data for that node. If there are n words in the language, then we can build a balanced binary search tree and so get lookup time of $O(\log n)$. But clearly some words appear far more often than others, so since the lookup time is the depth of the node, we want more frequently accessed items to be closer to the root (while still maintaining the fact that we have a binary search tree).

Let's formalize this a bit. Suppose we are given a sequence of n distinct keys $k_1 < k_2 < \cdots < k_n$, and for each i we are given a probability p_i that a search will be for k_i (so $\sum_{i=1}^{n} p_i = 1$). The book does a slightly more complicated analysis that allows searches for keys not in the input, but for now let's ignore this.

If we set for convention that the cost of a search is the number of nodes examined, and we define (as usual) the depth of a node to be the distance from the root, then clearly the cost of search for key k_i in tree T is $\text{depth}_T(k_i) + 1$. So for a binary search tree T on our input, the expected cost of a search is

$$\sum_{i=1}^{n} p_i(\text{depth}_T(k_i) + 1) = 1 + \sum_{i=1}^{n} p_i \cdot \text{depth}_T(k_i) \tag{12.3.2}$$

Our goal is to find the binary search tree T which minimizes this expected cost.

12.3.2 Dynamic programming algorithm

As always, we need to define the subproblems we consider and use an optimal substructure argument. This is pretty straightforward: suppose that k_r is the root. Then $k_1, \ldots, k_{r-1}$ are in the left subtree, and $k_{r+1}, \ldots, k_n$ are in the right subtree. And clearly the left subtree should be optimal for $k_1, \ldots, k_r$, while the right subtree should be optimal for $k_{r+1}, \ldots, k_n$ – if one of them was suboptimal, then we could simply use the optimal subtree instead and the expected cost would decrease.

Slightly more formally, for $1 \le i \le n$ and $i \le j \le n$, let $OPT(i, j)$ denote the binary search tree T on keys $k_i, \ldots, k_j$ which minimizes $c(T) = \sum_{a=i}^{j} p_a(\text{depth}_T(k_a)+1)$ (note that now the probabilities do not necessarily sum to 1). So we are looking for $OPT(1, n)$. By convention, we will say that $OPT(i, j)$ is the empty tree (no nodes) if $i < j$.

Theorem 12.3.1 *Let k_r be the root of $OPT(i, j)$. Then the left subtree of $OPT(i, j)$ is $OPT(i, r-1)$, and the right subtree of $OPT(i, j)$ is $OPT(r+1, j)$.*

Proof: Let $T = OPT(i, j)$, let T_L be the left subtree of T, and let T_R be the right subtree of T. Let T' be $OPT(i, r-1)$, and suppose that $T_L \neq T'$. Since T_L and T' are on the same set of keys and T' is optimal, by definition we know that $c(T') < c(T_L)$ (let's assume no ties for now). Let $\hat{T}$ denote the tree of $k_i, \ldots, k_j$ that we would get by replacing T_L with T'. Then

$$\begin{aligned}
c(\hat{T}) &= \sum_{a=i}^{j} p_a(\text{depth}_{\hat{T}}(k_a) + 1) \\
&= \sum_{a=i}^{r-1}(p_a(\text{depth}_{T'}(k_a) + 2)) + p_r + \sum_{a=r+1}^{j}(p_a(\text{depth}_{T_R}(k_a) + 2)) \\
&= \sum_{a=i}^{r-1}(p_a(\text{depth}_{T'}(k_a) + 1)) + \sum_{a=i}^{r-1} p_a + p_r + \sum_{a=r+1}^{j}(p_a(\text{depth}_{T_R}(k_a) + 1)) + \sum_{a=r+1}^{j} p_a \\
&= c(T') + \sum_{a=i}^{r-1} p_a + p_r + c(T_R) + \sum_{a=r+1}^{j} p_a
\end{aligned}$$

$$
\begin{aligned}
&< c(T_L) + \sum_{a=i}^{r-1} p_a + p_r + c(T_R) + \sum_{a=r+1}^{j} p_a \\
&= \sum_{a=i}^{r-1} (p_a(\text{depth}_{T_L}(k_a) + 1)) + \sum_{a=i}^{r-1} p_a + p_r + \sum_{a=r+1}^{j} (p_a(\text{depth}_{T_R}(k_a) + 1)) + \sum_{a=r+1}^{j} p_a \\
&= \sum_{a=i}^{r-1} (p_a(\text{depth}_{T_L}(k_a) + 2)) + p_r + \sum_{a=r+1}^{j} (p_a(\text{depth}_{T_R}(k_a) + 2)) \\
&= \sum_{a=i}^{j} p_a(\text{depth}_T(k_a) + 1) = c(T)
\end{aligned}
$$

This is a contradiction, since the fact that $T = OPT(i, j)$ means that $c(T) < c(\hat{T})$ by definition. Thus $T_L = T'$.

A symmetric argument works to prove that T_R must equal $OPT(r + 1, j)$. ■

Corollary 12.3.2 $c(OPT(i, j)) = \sum_{a=i}^{j} p_a + \min_{i \le r \le j}(c(OPT(i, r - 1)) + c(OPT(r + 1, j)))$

Proof: Let k_r be the root of $OPT(i, j)$. Then from Theorem 12.3.1, we know that the left subtree of $OPT(i, j)$ is $OPT(i, k - 1)$ and the right subtree of $OPT(i, j)$ is $OPT(r + 1, j)$. Thus the total cost of $OPT(i, j)$ is

$$
\begin{aligned}
c(OPT(i, j)) &= \sum_{a=i}^{j} p_a(\text{depth}_{OPT(i,j)}(k_a) + 1) \\
&= \sum_{a=i}^{r-1} (p_a(\text{depth}_{OPT(i,r-1)}(k_a) + 2)) + p_r + \sum_{a=r+1}^{j} p_a(\text{depth}_{OPT(r+1,j)}(k_a) + 2) \\
&= \sum_{a=i}^{j} p_a + \sum_{a=i}^{r-1} (p_a(\text{depth}_{OPT(i,r-1)}(k_a) + 1)) + \sum_{a=r+1}^{j} p_a(\text{depth}_{OPT(r+1,j)}(k_a) + 1) \\
&= \sum_{a=i}^{j} p_a + c(OPT(i, r - 1)) + c(OPT(r + 1, j)).
\end{aligned}
$$

On the other hand, by the same analysis, for any other tree for (i, j) rooted at r' the cost would be $\sum_{a=i}^{j} p_a + c(OPT(i, r' - 1)) + c(OPT(r' + 1, j))$. Thus $c(OPT(i, j)) = \sum_{a=i}^{j} p_a + \min_{i \le r \le j}(c(OPT(i, r - 1)) + c(OPT(r + 1, j)))$. ■

With this theorem in hand, it is now straightforward to give a recurrence for $M[i, j]$, which will be equal to $c(OPT(i, j))$. When computing $M[i, j]$, we can simply try all of the different possible roots and take the one with minimum cost, where we depend on Theorem 12.3.1 to use previously-computed costs of smaller trees.

$$
M[i, j] = \begin{cases} 0 & \text{if } i > j \\ \min_{i \le r \le j} \left(\sum_{a=i}^{j} p_a + M[i, r - 1] + M[r + 1, j] \right) & \text{if } i \le j \end{cases} \tag{12.3.3}
$$

Theorem 12.3.3 $M[i,j] = c(OPT(i,j))$

Proof: We prove this by induction on i and j (the partial order where $(i',j') < (i,j)$ iff $i' \leq i$ and $j' \leq j$). Note that while it's usual to do induction over a partial order, it's valid in this case (Google "well-founded induction" if you're not sure why). By definition, it is true when $i < j$. It is also true when $i = j$, since then the equation reduces to $M[i,j] = p_i$, and clearly $c(OPT(i,i)) = p_i$.

For the inductive step, consider an i and j with $i < j$. By the inductive hypothesis, $M[i',j'] = c(OPT(i',j'))$ for all i',j' with $i' \leq i$ and $j' \leq j$ (other than (i,j) itself). Suppose that k_r is the root of $OPT(i,j)$. When computing $M[i,j]$, by the definition of the algorithm we get a value of

$$\begin{aligned} M[i,j] &= \min_{i \leq r \leq j} \left(\sum_{a=i}^{j} p_a + M[i,r-1] + M[r+1,j] \right) \\ &= \min_{i \leq r \leq j} \left(\sum_{a=i}^{j} p_a + c(OPT(i,r-1)) + c(OPT(r+1,j)) \right) \qquad (12.3.4) \\ &= c(OPT(i,j)), \qquad (12.3.5) \end{aligned}$$

where in (12.3.4) we used the inductive hypothesis and in (12.3.5) we used Corollary 12.3.2. ■

As always, since the subproblems overlap it is not hard to show that if we do the naive recursive algorithm it will take exponential time. But if we do either bottom-up dynamic programming or top-down dynamic programming (memoization), we can avoid a lot of the recomputation and get the running time down to polynomial.

It's easy to see how to do this if we use memoization. For bottom-up, though, it's not clear what order we should iterate through the table. It's not hard to see that the obvious approaches (iterating through i and then j to compute $M[i,j]$ or vice versa) fail. So instead we'll do it in order of $j - i$. The details of this are in the book, but are also a good exercise.

What is the running time if we do dynamic programming? There are clearly n^2 table entries. To compute entry $M[i,j]$, we need to try all possible r between i and j, so the running time is $O(|j-i|) = O(n)$. Thus the overall running time is $O(n^3)$.