CS 3510 — Algorithms, Spring 2022

Lecture 9: More DP, Again

February 15, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

Warm Up Example

Let's say we have a $1 \times n$ plank that we want to cover with tiles. We have a total of 3 tiles available to us. We have a 1×1 red square, a 1×1 blue square, and a 1×2 red rectangle.

How many ways can we cover the $1 \times n$ plank? For this, both order and color matters. How can we solve this using dynamic programming? Let's say our table is $T[i]$ represents the number of ways to arrange tiles on a $1 \times i$ long plank.

The recurrence relation for $T[i]$ can be made by noting that if we have $i-2$ plank, we can make this is an i plank by placing a rectangle. Otherwise, if we have $i-1$ plank, we can place either a red or blue square. Hence, the recurrence relation is:

$$T[n] = T[n-2] + 2T[n-1]$$

and can be written with the following pseudocode:

```
1 Function PlankFill(n):
      /* let T be an empty n long list                    */
2     T ← []
3     T[1] ← 2
4     T[2] ← 5
5     for i ∈ 2 → n do
6         T[i] ← T[i − 1] + 2T[i − 2]
7     return T[n]
```

The runtime of this algorithm is $O(n)$ as to build any $T[i]$, we perform $O(1)$ work by looking at $T[i-1]$ and $T[i-2]$. Since we have a total of n entries to build, this takes $O(n)$ time.

Longest Common Subsequence

Recall that a subsequence can have gaps, as long as the order is consistent between the underlying array and the subsequence.

We are interested in the longest common subsequence between two lists A and B. In the example below, *a* common subsequence is $\{2, 9\}$

$A:$	1	$\boxed{2}$	5	$\boxed{9}$	3	8	7
$B:$	5	3	4	7	$\boxed{2}$	$\boxed{9}$	1

However, this is not the *longest* common subsequence, which is $\{5, 3, 7\}$.

$A:$	1	2	$\boxed{5}$	9	$\boxed{3}$	8	$\boxed{7}$
$B:$	$\boxed{5}$	$\boxed{3}$	4	$\boxed{7}$	2	9	1

Let us define $T[i]$ as the length of the LCS from $A[1:i]$ and $B[1:i]$, and compare A_i and B_i. We have to approach this using cases. We know that either A_i is equal to B_i or they are different.

If they are the same, then we can simply add this character to the LCS for the string not including A_i, B_i meaning we can write $T[i]$ as $T[i-1]+1$.

Another case: either A_i or B_i is not useful. If we include A_i and B_i, then this is not a common subsequence because these are obviously different.

If we pick to attach B_i, then the length of this LCS is the length of the LCS of $A[1:i-1]$ and $B[i:1]$. However, our table entry definition did not allow for the two strings to differ in length. This does not work.

Instead, it might make more sense to define our table entries as $T[i,j]$ where this is the length of the LCS between $A[1:i]$ and $B[1:j]$.

The case when A_i and B_j are the same, then $T[i,j] = 1 + T(i-1, j-1)$ which is the same as before.

When $A_i \neq B_j$, then we have that either A_i is useless to add, B_j is useless to add, or both A_i and B_j are useless to add. These refer to the LCS of $T(i-1,j), T(i,j-1), T(i-1,j-1)$.

Since we want the longest common subsequence, we can simply take the maximum of all these (since all these are solutions to subproblems).

Hence, in all, we can write the recurrence relation as:

$$T[i,j] = \begin{cases} T(i-1,j-1)+1 & A_i = B_i \\ \max\left(T(i,-1,j-1), T(i-1,j), T(i,j-1)\right) & A_i \neq B_i \end{cases}$$

Technically, the $T(i-1,j-1)$ is redundant because this will always be smaller than $T(i-1,j)$ or $T(i,j-1)$ and we don't need to include it.

We also have base cases of $T(0,j) = 0$ and $T(i,0) = 0$ for all i, j because there is no common subsequence for a string with zero length.

```
1  Function LCS(A, B):
       /* let T be an empty n+1 long list                    */
2      T ← []
3      for i ∈ 1 → n do
4          T[i, 0] ← 0
5      for j ∈ 1 → n do
6          T[0, j] ← 0
7      for i ∈ 1 → n do
8          for j ∈ 1 → n do
9              if A[i] = B[j] then
10                 T[i, j] ← 1 + T[i − 1, j − 1]
11             else
12                 T[i, j] = max(T(i − 1, j), T(i, j − 1))
13     return T[n, n]
```

The runtime of this algorithm is $O(n^2)$ as the base case takes $O(n)$ time to set and the nested for loop is $O(n^2)$ giving a total runtime of $O(n^2)$.

LCS for 3 Lists

How can we adapt the LCS for 3 input lists (A, B, C)? Can we simply say that the $LCS(A, B, C) = LCS(LCS(A, B), C)$? No! But, then how do we do it?

The table now holds values where $T[i, j, k]$ is the longest common subsequence between $A[1 : i], B[1 : i], C[1 : k]$ (the table is now three dimensional).

For the recurrence, notice that if $A_i = B_i = C_i$, then we can simply take $T[i, j, k] = 1 + T[i - 1, j - 1, k - 1]$. Otherwise, if they differ, we can simply take the max of all situations where we withhold a letter.

$$T[i, j, k] = \max\left(T[i - 1, j, k] + T[i, j - 1, k] + T[i, j, k - 1]\right)$$

We could also do $i - 1, j - 1, k$ or $i - 1, j, k - 1$ because those subproblems are strictly smaller than other subproblems already included, hence we can exclude those.

The runtime of this algorithm is $O(n^3)$ as we now have to traverse over the three dimensional table (rather than a 2D table).

Challenge

Let's start with an $n \times n$ matrix of integers and the goal is to get from the top left to the bottom right and we want the path with the smallest sum (we can only move right and down). Design a DP algorithm to solve this problem.