COMP 285 (NC A&T, Spr '22) Lecture 26

Dynamic Programmig IV - LCS, Unbounded Knapsack

1 Longest Common Subsequence

We now consider the longest common *subsequence* problem which has applications in spellchecking, biology (whether different DNA sequences correspond to the same protein), and more.

We say that a sequence Z is a *subsequence* of a sequence X if Z can be obtained from X by deleting symbols. For example, `abracadabra` has `baab` as a *subsequence*, because we can obtain `baab` by deleting a, r, cad, and ra. We say that a sequence Z is a longest common *subsequence* (LCS) of X and Y if Z is a *subsequence* of both X and Y , and any sequence longer than Z is not a *subsequence* of at least one of X or Y . For instance, the LCS of `abracadabra` and `bxqrabry` is `brabr`.

Using the definition of LCS, we define the LCS problem as follows: Given sequences X and Y , find the length of their LCS, Z (and if we are proceeding to Step 4 of the outline above, output Z). In what follows, suppose that the sequence X is $X = x_1x_2x_3 \cdots x_m$, so that X has length m, and suppose that $Y = y_1y_2 \cdots y_n$ as length n. We'll use the notation $X[1 : k]$ as usual to denote the prefix $X[1 : k] = x_1x_2 \cdots x_k$.

1.1 Steps 1 and 2: Identify optimal substructure, and write a recursive formulation

Our sub-problems will be to solve LCS on prefixes of X and Y. To see how we can do this, we consider the following two cases.

1. **Case 1:** $x_m = y_n$. If $x_m = y_n = \ell$, then any LCS Z has ℓ as its last symbol. Indeed, suppose that Z' is any common subsequence that does not end in ℓ: then we can always extend it by appending ℓ to Z' to obtain another (longer) legal common subsequence. Thus, if $|Z| = k$ and $x_m = y_n = \ell$, we can write

$$Z[1 : k - 1] = \text{LCS}(X[1 : m - 1], Y[1 : n - 1])$$

and

$$Z = Z[1 : k - 1] \circ \ell$$

, where $\circ$ denotes the concatenation operation on strings.

2. **Case 2:** $x_m \neq y_n$. As above, let Z be the LCS of X and Y . In this case, the last letter of Z (call it z_k) is either not equal to x_m or it is not equal to y_n. (Notice that this or

is not an exclusive or; maybe z_k isn't equal to either x_m or y_n). In this case, at least one of x_m or y_n cannot appear in the LCS of X and Y ; this means that either

$$\text{LCS}(X, Y) = \text{LCS}(X[1:m-1], Y)$$

or

$$\text{LCS}(X, Y) = \text{LCS}(X, Y[1:n-1])$$

, whichever is longer. That is, we can shave one letter off the end of either X or Y . In particular, the length of $\text{LCS}(X, Y)$ is given by

$$\text{lenLCS}(X, Y) = \max\{\text{lenLCS}(X[1:m-1], Y), \text{lenLCS}(X, Y[1:n-1])\}$$

.

This immediately gives us our recursive formulation. Let's keep a table C, so that

$$C[i, j] = \text{length of LCS}(X[1:i], Y[1:j])$$

Then we have the relationship:

$$C[i, j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & X[i] = Y[j], i, j > 0 \\ \max\{C[i-1, j], C[i, j-1]\} & X[i] \neq Y[j], i, j > 0 \end{cases}$$

Suppose we keep a table C, where $C[i, j]$ maintains the length of $\text{LCS}(X[1:i], Y[1:j])$, the longest common subsequence of $X[1:i]$ and $Y[1:j]$. Then, we can fill in the values of C using the following recurrence:

$$C[i, j] = \begin{cases} C[i-1, j-1] + 1 & X[i] = Y[j] \\ \max\{C[i-1, j], C[i, j-1]\} & \text{otherwise} \end{cases}$$

Technically, we should do a proof here to show that this recurrence is correct. See CLRS for the details, but it is true that if we define $C[i, j]$ recursively as above, then indeed, $C[i, j]$ is equal to the length of $\text{LCS}(X[1:i], Y[1:j])$. (Good exercise: prove this for yourself using induction).

1.2 Step 3: Define an algorithm using our recursive relationship.

The recursive relationship above naturally gives rise to a DP algorithm for filling out the table C:

Note that there are only $n \times m$ entries in our table C. This is where the overlapping subproblems come in: we only need to compute each entry once, even though we may access it many times when filling out subsequent entries.

Algorithm 1: lenLCS(X,Y)

```
Initialize an $n+1 \times m+1$ zero-indexed array $C$
Set $C[0,j] = C[i,0] = 0$ for all $i, j \in \{1, \cdots, m\} \times \{1, \cdots, n\}$
for $i = 1, \cdots, m$ do
  for $j = 1, \cdots, n$ do
    if $X[i] == Y[j]$ then
      $C[i,j] = 1 + C[i-1, j-1]$
    else
      $C[i,j] = \max\{C[i-1,j], C[i,j-1]\}$
    end if
  end for
end for
return C
```

We can also see that $C[i,j]$ only depends on three possible prior values: $C[i-1,j], C[i,j-1]$, and $C[i-1,j-1]$. This means that each time we compute a new value $C[i,j]$ from previous entries, it takes time $O(1)$.

Thus, we can start to see how to obtain an algorithm for filling in the table and obtaining the LCS. First, we know that any string of length 0 will have an LCS of length 0. Thus, we can start by filling out $C[0,j] = 0$ for all j and similarly, $C[i,0] = 0$ for all i. Then, we can fill out the rest of the table, filling the rows from bottom up (i from 1 to m) and filling each row from left to right (j from 1 to n). The pseudocode is given in Algorithm 1.

As mentioned above, in order to fill each entry, we only need to perform a constant number of lookups and additions. Thus, we need to do a constant amount of work for each of the $m \times n$ entries, giving a running time of $O(mn)$.

1.3 Step 4: Recovering the actual LCS

Algorithm 1 only computes the length of the LCS of X and Y . What if we want to recover the actual longest common subsequence? In Algorithm 2, we show how we can construct the actual LCS, given the dynamic programming table C that we've filled out in Algorithm 1.

In this algorithm, we start from the end of X and Y and work backward, using our table C as a guide. We start with $i = m$ and $j = m$. If at some point (i,j), we see that $X[i] = Y[j]$, then decrement both i and j. On the other hand, if $X[i] \neq Y[j]$, then we know that we need to drop a symbol from either X or Y . The table C will tell us which: if $C[i,j] = C[i,j-1]$, then we can drop a symbol from Y and decrement j. If $C[i,j] = C[i-1,j]$, then we can drop a symbol from X and decrement i. Of course, it might be the case that both of these hold; in this case it doesn't matter which we decrement, and our pseudocode will be default decrement j.

How long does this take? Notice that in each step, the sum $i + j$ is decremented by at least one (maybe two) and stops as soon as one of i, j is equal to zero; this is at least before $i + j = 0$. Thus, the number of times we decrement $i + j$ is at most $m + n$, which was their total value to start.

Because at each step of Algorithm 2, the work is $O(1)$, the total running time is thus $O(n+m)$, which is subsumed by the runtime of $O(mn)$ necessary to fill in the table.

Algorithm 2: LCS(X,Y)

```
// C is filled out already
$L \leftarrow$
$i \leftarrow m$
$j \leftarrow n$
while $i > 0$ and $j > 0$ do
  if $X[i] = Y[j]$ then
    Append $X[i]$ to the beginning of $L$
    $i \leftarrow i - 1$
    $j \leftarrow j - 1$
  else if $C[i, j] = C[i, j - 1]$ then
    $j \leftarrow j - 1$
  else
    $i \leftarrow i - 1$
  end if
end while
```

The conclusion is that we can find LCS(X, Y) of a sequence X of length m and a sequence Y of length n in time $O(mn)$.

Interestingly, this simple dynamic programming algorithm is basically the best known algorithm or solving the LCS problem. It is conjectured that this algorithm may be essentially optimal. It turns out that giving an algorithm that (polynomially) improves the dependence on m andn over the $O(mn)$ strategy outlined above would imply a major breakthrough in algorithmsfor the boolean satisfiability problem – a problem widely believed to be computationally hard to solve.

2 The Knapsack Problem

This is a classic problem, defined as the following:

We have n items, each with a value and a positive weight. The i-th item has weight w_i and value v_i. We have a knapsack that holds a maximum weight of W. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

Item	Weight	Value
A	6	25
B	3	13
C	4	15
D	2	8

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

2.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item B and two of item D. Then, the total weight is 10, and the total value 42.

We define $K(x)$ to be the optimal solution for a knapsack of capacity x. Suppose $K(x)$ happens to contain the i-th item. Then, the remaining items in the knapsack must have a total weight of at most $x - w_i$. The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

$$K(x) = \max_{i:w_i \leq x} (K(x - w_i) + v_i)$$

.

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize $K(0) = 0$, and then we compute $K(x)$ values from $x = 1, \cdots, W$. The overall runtime is $O(nW)$.

Remark 1. *This solution is not actually polynomial in the input size because it takes* $\log W$) *bits to represent W. We call these algorithms "pseudo-polynomial." If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.*

Algorithm 3: UnboundedKnapsack(W, n, w, v)

```
$K[0] \leftarrow 0$
**for** $x = 1, \cdots, W$ **do**
  $K[x] \leftarrow 0$
  **for** $i = 1, \cdots, n$ **do**
    **if** $w_i <= x$ **then**
      $K[x] \leftarrow \max\{K[x - w_i] + v_i\}$
    **end if**
  **end for**
**end for**
**return** $K[W]$
```