CS 31: Algorithms (Spring 2019): Lecture 8

Date: 11th April, 2019

Topic: Dynamic Programming 3: Manipulating Strings

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

In this lecture, we will look at two or three "string" problems. The input to these problems will be strings of the form $s[1:n]$ where each $s[i]$ will be from some alphabet Σ; the alphabet could be $\{0,1\}$, the Roman alphabet, or $\{A,C,G,T\}$, etc, etc.

1 Longest Common Subsequence (LCS)

Given a string $s[1:n]$ a *subsequence* is a subset of the various "coordinates" in the same order as the string. Formally, a length k subsequence is a string $\sigma = (s[i_1] \circ s[i_2] \circ \ldots \circ s_[i_k])$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. For example, if the string is `algorithms`, of length 10, then `lot` is a subsequence with $i_1 = 2, i_2 = 4$, and $i_3 = 7$. Similarly, `grim` is a subsequence. But, `list` is not a subsequence.

Remark: *Note that the $i_1, i_2, \ldots, i_k$ need not be contiguous; if they are indeed contiguous, then the subsequence is called a* substring. *The number of substrings are at most $O(n^2)$; the number of subsequences can be $O(2^n)$.*

Given two strings $s[1:m]$ and $t[1:n]$, a string σ is a *common subsequence* if it appears in both as a subsequence. Formally, if $|\sigma| = k$ then there exists $(i_1 < \ldots < i_k)$ and $(j_1 < \ldots < j_k)$, such that $s[i_r] = t[j_r]$ for all $1 \leq r \leq k$. Once again, the locations don't need to be the same, that is, i_r needn't be j_r. For example, if $s =$ `algorithms` and $t =$ `computers`, then the string $\sigma =$ `oms` is a subsequence with $(i_1, i_2, i_3) = (4, 9, 10)$ and $(j_1, j_2, j_3) = (2, 3, 9)$.

LONGEST COMMON SUBSEQUENCE
Input: Two strings $s[1:m]$ and $t[1:n]$.
Output: Return a longest common subsequence between s and t.
Size: m, n.

As remarked before the problem definition, the number of subsequences can be exponentially many and brute-forcing over them is not a great idea. Instead, we imagine the longest common subsequence σ^* of s and t. And as we did for subset-sum and knapsack, we consider the "last" element of σ^*.

Suppose $|\sigma^*| = k$, and for the time-being suppose we just want this value k. We will recover the solution later as we did for Subset Sum and Knapsack. Let us try to (informally) observe how $|\sigma^*|$ can be argued about as arising out of solutions to smaller instances of the same problem.

- *Case 1:* $\sigma^*[k] \neq s[m]$ and $\sigma^*[k] \neq t[n]$. This happens, for example, when s is `apple` and t is `apply` and σ^* is `appl`. We observe that σ^* should be the LCS of $s[1:m-1]$ and $t[1:n-1]$.
- *Case 2:* $\sigma^*[k] = s[m]$ and $\sigma^*[k] \neq t[n]$. This happens, for example, when s is `appal` and t is `apply` and σ^* is `appl`. We observe that σ^* should be the LCS of $s[1:m]$ and $t[1:n-1]$.
- *Case 3:* $\sigma^*[k] \neq s[m]$ and $\sigma^*[k] = t[n]$. This is absolutely symmetric to Case 2: in this case σ^* is the LCS of $s[1:m-1]$ and $t[1:n]$.
- *Case 4:* $\sigma^*[k] = s[m]$ and $\sigma^*[k] = t[n]$. This happens, for example, if s is `appal` and t is `appeal` and σ^* is `appl`. We observe that $\sigma^*[1:k-1]$ should then be the LCS of $s[1:m-1]$ and $t[1:n-1]$, and then we append the `l` to this answer.

Therefore, we see that if some one had given us the solution to the following 3 smaller instances of LCS namely the LCS between $(s[1:m-1], t[1:n-1])$, and $(s[1:m], t[1:n-1])$, and $(s[1:m-1], t[1:n])$, then we can get the LCS of $(s[1:m], t[1:n])$. We observe that the lengths of the input strings drop, and in particular, any one of them could drop. We extrapolate from here that the smaller instances that we would eventually need will be of the LCS of $s[1:i]$ and $s[1:j]$ for $1 \leq i \leq m$ and $1 \leq i \leq m$; we should store these solutions in a 2-dimensional array $T[i,j]$. Finally, the base case: when $i = 0$ or $j = 0$, the length of the LCS will also be 0.

We have all the ingredients for the dynamic programming solution which we now provide below.

1. *Definition:* For any $0 \leq i \leq m$ and $0 \leq j \leq n$, let us use $\mathsf{LCS}(i,j)$ to be the length of the longest common subsequence of $s[1:i]$ and $t[1:j]$. We are interested in $\mathsf{LCS}(m,n)$.

   As in the case of knapsack, it is useful to introduce the notation of $\mathsf{Cand}(i,j)$. Let $\mathsf{Cand}(i,j)$ to be the set of all *common subsequences* of the strings $s[1:i]$ and $t[1:j]$. With this notation, we get

   $$\mathsf{LCS}(i,j) = \max_{\sigma \in \mathsf{Cand}(i,j)} |\sigma|$$

2. *Base Cases:* $\mathsf{LCS}(0,j) = 0$ for all $0 \leq j \leq n$ and $\mathsf{LCS}(i,0) = 0$ for all $0 \leq i \leq m$.

3. *Recursive Formulation:* Let $\mathbf{1}_{i,j}$ be the indicator variable defined as

   $$\mathbf{1}_{i,j} = \begin{cases} 1 & \text{if } s[i] = t[j] \\ 0 & \text{otherwise} \end{cases}$$

   For all $i > 0, j > 0$:

   $$\mathsf{LCS}[i,j] = \max(\ \mathsf{LCS}[i-1,j],\ \mathsf{LCS}[i,j-1],\ \mathsf{LCS}[i-1,j-1] + \mathbf{1}_{i,j}\ )$$

4. *Formal Proof:*

   ($\geq$): Let σ be the subsequence in $\mathsf{Cand}(i-1,j)$ of length $\mathsf{LCS}(i-1,j)$. Since $\mathsf{Cand}(i-1,j) \subseteq \mathsf{Cand}(i,j)$, we get $\mathsf{LCS}(i,j) \geq \mathsf{LCS}(i-1,j)$ since the former maximizes over a larger set. Similarly, $\mathsf{LCS}(i,j) \geq \mathsf{LCS}(i,j-1)$. Finally, we note if $\sigma' \in \mathsf{Cand}(i-1,j-1)$ *and* $s[i] = t[j]$, then $\sigma' \circ s[i]$ is a common subsequence in $\mathsf{Cand}(i,j)$. This implies, $\mathsf{LCS}(i,j) \geq \mathsf{LCS}(i-1,j-1) + \mathbf{1}_{i,j}$.

   ($\leq$): Let σ^* be the subsequence in $\mathsf{Cand}(i,j)$ of length $\mathsf{LCS}(i,j)$. Let $k = |\sigma^*|$. Now repeat the arguments in the 4 cases above.

   - *Case 1:* $\sigma^*[k] \neq s[i]$ and $\sigma^*[k] \neq t[j]$. Then $\sigma^* \in \mathsf{Cand}(i-1,j-1)$. Therefore, $\mathsf{LCS}(i,j) \leq \mathsf{LCS}(i-1,j-1) = \mathsf{LCS}(i-1,j-1) + \mathbf{1}_{i,j}$.
   - *Case 2:* $\sigma^*[k] = s[i]$ and $\sigma^*[k] \neq t[j]$. Then $\sigma^* \in \mathsf{Cand}(i,j-1)$ and so $\mathsf{LCS}(i,j) \leq \mathsf{LCS}(i,j-1)$.
   - *Case 3:* $\sigma^*[k] \neq s[i]$ and $\sigma^*[k] = t[j]$. Then $\sigma^* \in \mathsf{Cand}(i-1,j)$ and so $\mathsf{LCS}(i,j) \leq \mathsf{LCS}(i-1,j)$.
   - *Case 4:* $\sigma^*[k] = s[m]$ and $\sigma^*[k] = t[n]$. Then $\sigma^* - \sigma[k] \in \mathsf{Cand}(i-1,j-1)$, and $\mathbf{1}_{i,j} = 1$. Therefore, $\mathsf{LCS}(i,j) - 1 \leq \mathsf{LCS}(i-1,j-1)$, implying $\mathsf{LCS}(i,j) \leq \mathsf{LCS}(i-1,j-1) + \mathbf{1}_{i,j}$.

   In each case, $\mathsf{LCS}(i,j)$ is less than one of the three things in the RHS.

5. *Pseudocode for computing* $\mathsf{LCS}[m,n]$ *and recovery pseudocode:*

```
1:  procedure LCS(s[1 : m], t[1 : n]):
2:      ▷ Returns the longest common subsequence of s and t.
3:      Allocate space L[0 : m, 0 : n] ▷ L[i, j] will contain the length of the LCS of
        s[1 : i] and t[1 : j].
4:      L[0, j] ← 0 for all 0 ≤ j ≤ n and L[i, 0] ← 0 for all 0 ≤ i ≤ m. ▷ Base
        Cases.
5:      for 1 ≤ i ≤ m do:
6:          for 1 ≤ j ≤ n do:
7:              L[i, j] ← max( L[i − 1, j], L[i, j − 1], L[i − 1, j − 1] + 1_{i,j} )
8:      ▷ L[m, n] now contains the value of the longest common subsequence
9:      ▷ Below we show the recovery pseudocode

10:     i ← m; j ← n; σ = [].
11:     ▷ Invariant: |σ| + L[i, j] = L[m, n]
12:     while i > 0 and j > 0 do:
13:         if L[i, j] = L[i − 1, j − 1] + 1_{i,j} then:
14:             if 1_{i,j} = 1 then:
15:                 Append s[i] to the front of σ.
16:             i ← i − 1; j ← j − 1
17:         else if L[i, j] = L[i − 1, j] then:
18:             i ← i − 1
19:         else: ▷ We must have that L[i, j] = L[i, j − 1]
20:             j ← j − 1
21:     return σ
```

Note that in the recovery the invariant always holds and at the end since $L[0, j] = 0$ or $L[i, 0] = 0$, we have $|\sigma| = L[m, n]$.

6. *Running time and space* The above pseudocode take $O(mn)$ time and space.

Theorem 1. The LONGEST COMMON SUBSEQUENCE between two strings can be found in $O(nm)$ time and space.

2 Edit Distance

Remark: *Unfortunately, we didn't have time to go over this in class. We just remarked how you would use the* ***same*** *idea as in LCS, almost word to word, to compute the edit-distance. Please try yourself first, and then try to solve the remaining.*

This is a similar problem to the longest common subsequence problem. Given two strings $s[1 : m]$ and $t[1 : n]$, the *edit distance* is notion of distance between s and t defined

using 3 operations. The first is the *insert* operation – $\mathsf{ins}(s, i, c)$ inserts character c between $s[i]$ and $s[i+1]$, thus making s longer; $\mathsf{del}(t, j)$ deletes $t[j]$ from t making it shorter; and $\mathsf{sub}(s, i, c)$ replaces $s[i]$ with the character c keeping the length the same. Each operation costs 1 unit.

The edit distance between $s[1:m]$ and $t[1:n]$ is the minimum number of operations above that are required to convert s into t. This is denoted as $\mathsf{ED}(s, t)$.

For example, if s is `apple` and t is `banana`, then $\mathsf{ED}(s, t) = 5$ since one can go from `apple` $\rightarrow$ `bapple` $\rightarrow$ `banple` $\rightarrow$ `banale` $\rightarrow$ `banane` $\rightarrow$ `banana`. The operations are $\mathsf{ins}(s, 1, b)$, $\mathsf{sub}(s, 3, n)$, $\mathsf{sub}(s, 4, a)$, $\mathsf{sub}(s, 5, n)$, and $\mathsf{sub}(s, 6, e)$.

EDIT DISTANCE
Input: Two strings $s[1:m]$ and $t[1:n]$.
Output: Return $\mathsf{ED}(s, t)$.
Size: m, n.

✍

Exercise:

- *Prove that* $\mathsf{ED}(s, t) = \mathsf{ED}(t, s)$.
- *Prove that for any three strings* s, t, u*, we have* $\mathsf{ED}(s, u) \leq \mathsf{ED}(s, t) + \mathsf{ED}(t, u)$.

The edit distance can be computed by almost the same algorithm as above for LCS.

1. *Definition:* For any $0 \leq i \leq m$ and $0 \leq j \leq n$, let us use $\mathsf{ED}(i, j)$ to be the edit distance between the strings $s[1:i]$ and $t[1:j]$. We are interested in $\mathsf{ED}(m, n)$.

   What should $\mathsf{Cand}(i, j)$ be? Since the edit distance is the smallest number of "string operations" (ins/del/sub), let's define $\mathsf{Cand}(i, j)$ as the all possible sequences π of string operations which take $s[1:i]$ to $t[1:j]$. Armed with this notation, we get

   $$\mathsf{ED}(i, j) = \min_{\pi \in \mathsf{Cand}(i,j)} |\pi|$$

2. *Base Cases:*

   $\mathsf{LCS}(0, j) = j$ for all $0 \leq j \leq n$ and $\mathsf{LCS}(i, 0) = i$ for all $0 \leq i \leq m$. There is only one way to go from an empty string to a string of length i or j – keep inserting.

3. *Recursive Formulation:* As before, let $\mathbf{1}_{i,j}$ be the indicator variable defined as

   $$\mathbf{1}_{i,j} = \begin{cases} 1 & \text{if } s[i] = t[j] \\ 0 & \text{otherwise} \end{cases}$$

   For all $i > 0, j > 0$:

   $$\mathsf{ED}[i, j] = \min(\ 1 + \mathsf{ED}[i-1, j],\ 1 + \mathsf{ED}[i, j-1],\ (1 - \mathbf{1}_{i,j}) + \mathsf{ED}[i-1, j-1]\ )$$

4. *Formal Proof:*

   ($\leq$): Let π be the sequence of operations in $\mathsf{Cand}(i-1, j)$ of length $\mathsf{ED}(i-1, j)$. Consider the sequence of operations $\pi' = \mathsf{del}(s, s[i], i) \circ \pi$, which first *deletes* the last entry of $s[1:i]$ togo to $s[1:i-1]$, and then follows the sequence of operations in π to get to $s[1:j]$. Thus, $\pi' \subseteq \mathsf{Cand}(i, j)$ and $|\pi'| = 1 + |\pi| = 1 + \mathsf{ED}(i, j)$. Therefore, we get $\mathsf{ED}(i, j) \leq 1 + \mathsf{ED}(i-1, j)$ since the former is $\min_{\pi \in \mathsf{Cand}(i,j)} |\pi|$.

   Similarly, one can show $\mathsf{ED}(i, j) \leq 1 + \mathsf{ED}(i, j-1)$; the only difference is that we would $\mathsf{ins}(s[1:j-1], s[j], j)$ at the end of doing π. (You should try writing this yourself).

   Finally, suppose π was a sequence of operations that took $s[1:i-1]$ to $t[1:j-1]$ and whose length was $\mathsf{ED}(i-1, j-1)$. If $s[i] = t[j]$, then π also takes $s[1:i]$ to $t[1:j]$. If $s[i] \neq t[j]$, then consider the sequence $\pi' = \mathsf{sub}(s, t[j], i) \circ \pi$; this takes $s[1:i]$ to $t[i:j]$. Note that $|\pi'| = (1 - \mathbf{1}_{i,j}) + |\pi| = (1 - \mathbf{1}_{i,j}) + \mathsf{ED}(i-1, j-1)$.

   ($\geq$): Let π^* be the sequence of operations which took $s[1:i]$ to $t[1:j]$.

   Look at all operations of π^* which don't involve the position $s[i]$. The remaining operations must be taking $s[1:i-1]$ to $t[1:j]$. And after that, the position $s[i]$ must be deleted in π^*. Thus, there is a sequence π of operations of length $|\pi| = |\pi^*| - 1 = \mathsf{ED}(i, j) - 1$ which takes $s[1:i-1]$ to $t[1:j]$. Thus, $\mathsf{ED}(i-1, j) \leq \mathsf{ED}(i, j) - 1$. Similarly arguing, we get $\mathsf{ED}(i, j-1) \leq \mathsf{ED}(i, j) - 1$.

   Now, look at all the operations of π^* which don't involve $s[i]$ or $t[j]$. They must take the string $s[1:i-1]$ to $t[1:j-1]$. In particular, there is a sequence of operations π with $|\pi| \leq |\pi^*|$ which lies in $\mathsf{Cand}(i-1, j-1)$. Thus, $\mathsf{ED}(i, j) \geq \mathsf{ED}(i-1, j-1)$. Furthermore, if $s[i] \neq t[j]$, then need to be substituted after $s[1:i-1]$ is taken to $t[1:j-1]$. Thus, $\mathsf{ED}(i, j) \geq \mathsf{ED}(i-1, j-1) + (1 - \mathbf{1}_{i,j})$.

5. *Pseudocode for computing* $\mathsf{ED}[m, n]$.

```
1: procedure ED(s[1:m], t[1:n]):
2:     ▷ Returns the edit distance between s and t.
3:     Allocate space E[0:m, 0:n] ▷ E[i, j] will contain the edit distance between
   s[1:i] and t[1:j].
4:     E[0, j] ← j for all 0 ≤ j ≤ n and E[i, 0] ← i for all 0 ≤ i ≤ m. ▷ Base
   Cases.
5:     for 1 ≤ i ≤ m do:
6:         for 1 ≤ j ≤ n do:
7:             E[i, j] ← min( E[i−1, j], E[i, j−1], E[i−1, j−1] + (1 − 1_{i,j}) )
8:     return E[m, n].
```

6. *Running time and space* The above pseudocode take $O(mn)$ time and space.

Theorem 2. The EDIT DISTANCE between two strings can be found in $O(nm)$ time and space.

✍

Exercise: *Write the recovery pseudocode, that is, which gives the sequence of operations which take $s[1:n]$ to $t[1:m]$. To really appreciate it, you do need to code it and see how to get from* `apple` *to* `banana`.