CS31 (Algorithms), Spring 2020 : Lecture 8

Date:

Topic: Dynamic Programming 3: String Problems

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

---

In this lecture, we will look at another type of poster-child problems for dynamic programming: "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 $\{A,C,G,T\}$, depending on the applications.

1 Longest Common Subsequence (LCS)

Given a string $s[1:n]$ a *subsequence* is a subset of 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 $1 \le i_1 < i_2 < \cdots < i_k \le n$. 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 $O(2^n)$ (do you see this?).*

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 $1 \le r \le k$. 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. Once again, the idea from dynamic programming will give a much more efficient algorithm.

As in SUBSET SUM and KNAPSACK, we imagine the longest common subsequence σ^* of s and t. Suppose $|\sigma^*| = k$, and for the time-being suppose we just want this value k. We will recover the actual subsequence later. Let us try to get "solutionettes" from σ^* by considering the "last" element of $\sigma^*[k]$. We assert the following (all assertions will be proved formally later).

- *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`. In this case σ^* itself is 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`. In this case σ^* is 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`. In this case $\sigma^*[1:k-1]$ is the LCS of $s[1:m-1]$ and $t[1:n-1]$.

Therefore, from the instance $I = (s[1:m], t[1:n])$, we get three smaller subinstances $I_1 = (s[1:m-1], t[1:n-1])$, $I_2 = (s[1:m], t[1:n-1])$, and $I_3 = (s[1:m-1], t[1:n])$, and it seems that given the solutions to I_1, I_2, I_3 one can obtain the solution to I (once again, formal proof later). If one were to draw the recursion tree more, one would observe that a "typical subinstance'; would look like $I' = (s[1:i], t[1:j])$. Thus, these are parametrized by $0 \leq i \leq m$ and $0 \leq j \leq n$, and so can be arranged in an $(m+1) \times (n+1)$ grid. The base case: when $i = 0$ or $j = 0$, that is when one of the strings is empty, the length of the LCS will also be 0. We have all the ingredients for the dynamic programming solution which we now rigorously 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 $\mathsf{LCS}(m,n)$.

   Since this is an optimization problem, 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 super-set. Similarly, $\mathsf{LCS}(i,j) \geq \mathsf{LCS}(i,j-1)$ and $\mathsf{LCS}(i,j) \geq \mathsf{LCS}(i-1,j-1)$. 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:***

```
procedure LCS($s[1:m], t[1:n]$):
    ▷ Returns the longest common subsequence of s and t.
    Allocate space $L[0:m, 0:n]$ ▷ $L[i,j]$ will contain the length of the LCS of $s[1:i]$ and $t[1:j]$.
    $L[0,j] \leftarrow 0$ for all $0 \leq j \leq n$ and $L[i,0] \leftarrow 0$ for all $0 \leq i \leq m$. ▷ Base Cases.
    for $1 \leq i \leq m$ do:
        for $1 \leq j \leq n$ do:
            $L[i,j] \leftarrow \max(\ L[i-1,j],\ L[i,j-1],\ L[i-1,j-1] + \mathbf{1}_{i,j}\ )$
    ▷ $L[m,n]$ now contains the value of the longest common subsequence
    ▷ Below we show the recovery pseudocode
    $i \leftarrow m; j \leftarrow n; \sigma = []$.
    ▷ Invariant: $|\sigma| + L[i,j] = L[m,n]$
    while $i > 0$ and $j > 0$ do:
        if $L[i,j] = L[i-1,j-1] + \mathbf{1}_{i,j}$ then:
            if $\mathbf{1}_{i,j} = 1$ then:
                Append $s[i]$ to the front of $\sigma$.
                ▷ We are forming $\sigma$ from right to left.
            $i \leftarrow i-1; j \leftarrow j-1$
        else if $L[i,j] = L[i-1,j]$ then:
            $i \leftarrow i-1$
        else: ▷ We must have that $L[i,j] = L[i,j-1]$
            $j \leftarrow j-1$
    return $\sigma$
```

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

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, $\mathsf{ins}(s,i,c)$, which inserts character c between $s[i-1]$ and $s[i]$, thus making s longer; $\mathsf{del}(s,j)$ deletes $s[j]$ from s 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 $\mathsf{ED}(s,t) \le 5$ since one can go from `apple` → `bapple` → `banple` → `banale` → `banane` → `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 $\mathsf{sub}(s,6,a)$.

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

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

1. ***Definition:*** For any $0 \le i \le m$ and $0 \le j \le n$, 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:***

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

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:***

   ($\le$): 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 $\pi' = \mathsf{del}(s,i) \circ \pi$, which first *deletes* the last entry of $s[1:i]$ to get $s[1:i-1]$, 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 $\mathsf{ED}(i,j) \le 1 + \mathsf{ED}(i-1,j)$ since the former is $\min_{\pi \in \mathsf{Cand}(i,j)} |\pi|$. Similarly, one can show $\mathsf{ED}(i,j) \le 1 + \mathsf{ED}(i,j-1)$; the only difference is that we would $\mathsf{ins}(t[1:j-1], t[j], j)$ at the end of doing π.

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 $s[i] \ne t[j]$, then consider the sequence $\pi' = \mathsf{sub}(s, t[j], i) \circ \pi$; this takes $s[1:i]$ to $t[1:j]$. Note that $|\pi'| = (1 - \mathbf{1}_{i,j}) + |\pi| = (1 - \mathbf{1}_{i,j}) + \mathsf{ED}(i-1,j-1)$.

($\ge$): Let π^* be the sequence of operations which took $s[1:i]$ to $t[1:j]$. Note that, in π^*, either $s[i]$ is deleted from the end of string, or $t[j]$ is inserted to the end of the string, and if neither of these two occur, we must either substitute $s[i]$ and $t[j]$, or they are the same. In the first case, consider the sequence of operations π which is π^* without the deletion. Observe, that π acting on $s[1:i-1]$ would take us to $t[1:j]$. Thus, $|\pi^*| = 1 + |\pi| \ge 1 + \mathsf{ED}(i-1,j)$. Similarly, in the second case, consider the sequence π which is π^* without the insertion. π takes us from $s[1:i]$ to $t[1:j-1]$, and thus, in this case, $|\pi^*| \ge 1 + \mathsf{ED}(i,j-1)$. Finally, if neither of the above two occur, then either $s[i] = t[j]$ in which case π^* actually takes $s[1:i-1]$ to $t[1:j-1]$. That is, $\pi^* \ge \mathsf{ED}(i-1,j-1) = (1 - \mathbf{1}_{i,j}) + \mathsf{ED}(i-1,j-1)$ since $s[i] = t[j]$. Or, $s[i] \ne t[j]$, and there is a substitution. And in this case, π defined as π^* minus that substitution takes $s[1:i-1]$ to $t[1:j-1]$. Again giving, $\pi^* \ge 1 + \mathsf{ED}(i-1,j-1) = (1 - \mathbf{1}_{i,j}) + \mathsf{ED}(i-1,j-1)$ in this case. In sum, in all of the possible cases, $\mathsf{ED}(i,j) = |\pi^*|$ is larger than one of the things in the RHS paranthesis.

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:     $E[0,j] \leftarrow j$ for all $0 \le j \le n$ and $E[i,0] \leftarrow i$ for all $0 \le i \le m$. ▷ Base Cases.
5:     for $1 \le i \le m$ do:
6:         for $1 \le j \le n$ do:
7:             $E[i,j] \leftarrow \min(\ E[i-1,j],\ E[i,j-1],\ E[i-1,j-1] + (1 - \mathbf{1}_{i,j})\ )$
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 $t[1:m]$. Run it to see how to get from* `apple` *to* `banana`.